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

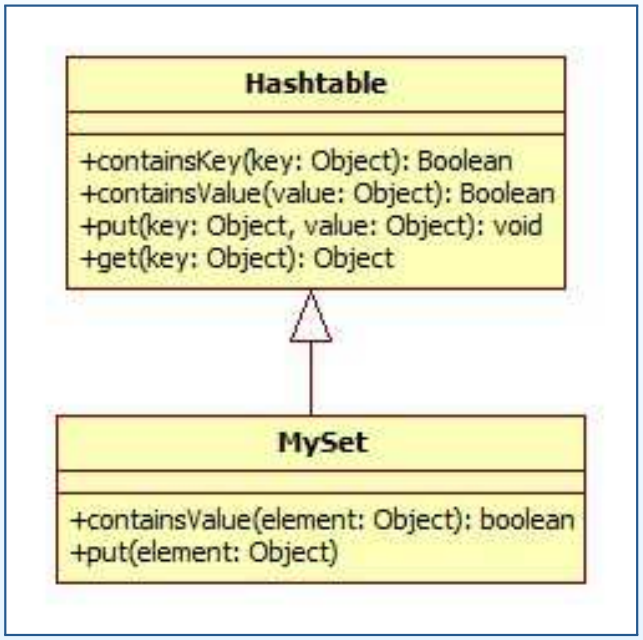
- Diagramele UML de clase pot fi utilizate pentru modelarea atât a domeniului problemei, cât şi a domeniului soluţiei
- Dualitatea *domeniul problemei* / *domeniul soluţiei*
  - *Obiectele din domeniul problemei* (eng. *domain objects*, *application objects*) corespund unor concepte ale domeniului relevante 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

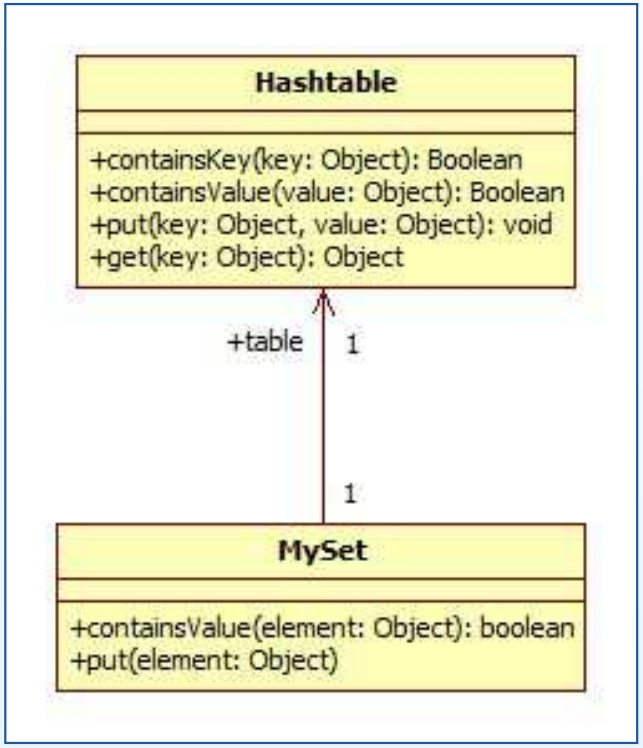
- *Moştenirea implementării* (eng. *implementation inheritance*) = utilizarea moştenirii având drept unic scop reutilizarea codului
- *Moştenirea specificării* sau *moştenirea interfeţei* (eng. *specification inheritance*, *interface inheritance*) = clasificarea conceptelor în ierarhii

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

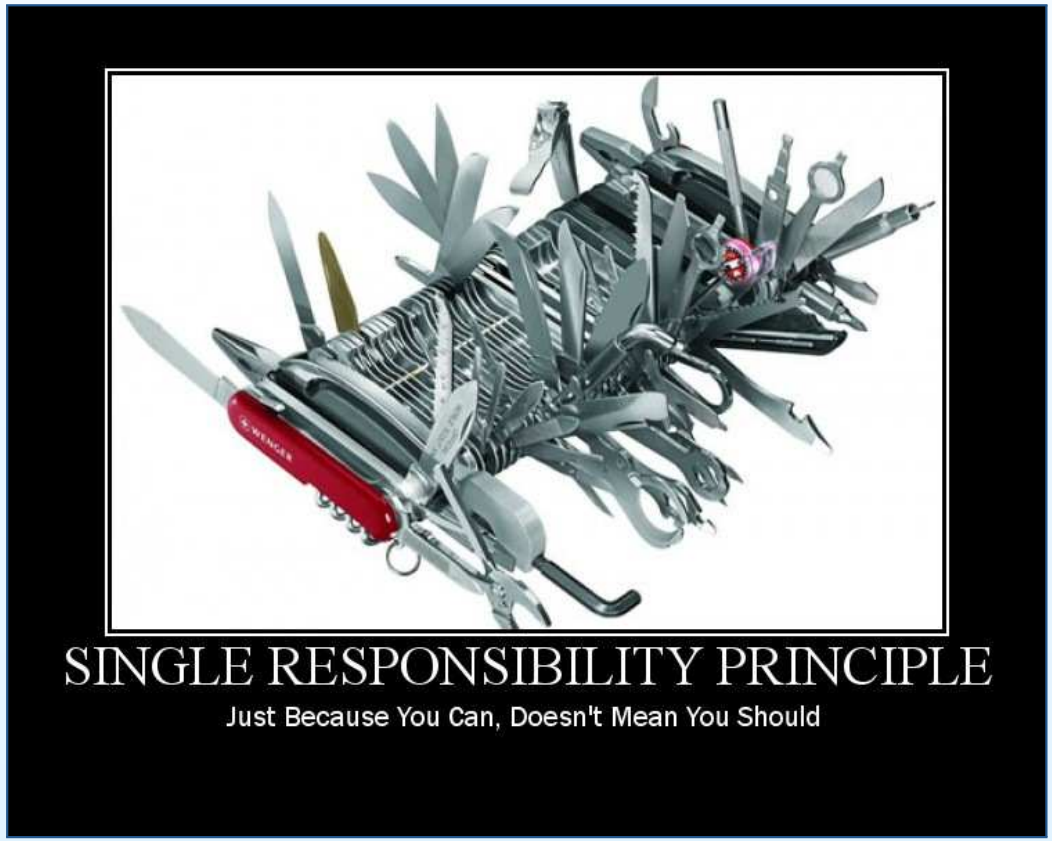
- Oferă o definiţie formală conceptului de *moştenire a specificării*
- (B. Liskov, 1988) Dacă pentru fiecare obiect *o1* de tipul *S* există un obiect *o2* de tipul *T*, astfel încât orice program *P* definit în termenii lui *T* îşi păstrează comportamentul atunci când *o2* este substituit cu *o1*, 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

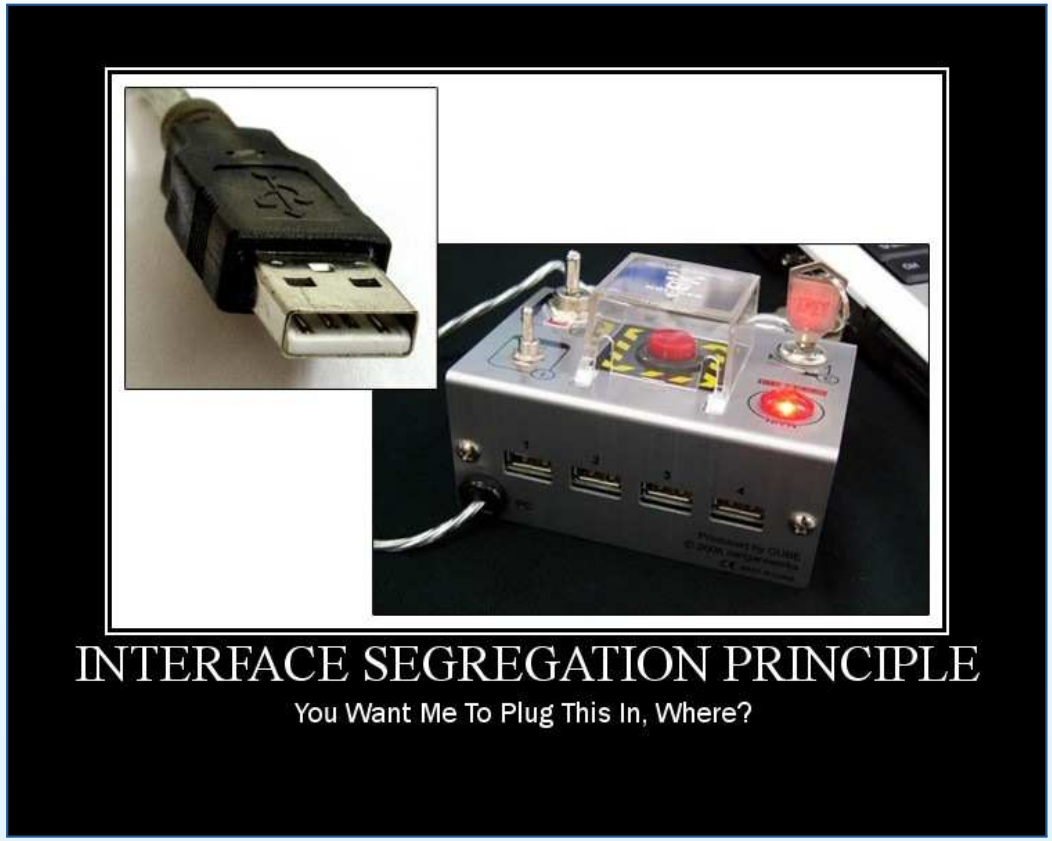


Principiul deschis/închis

- 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*)

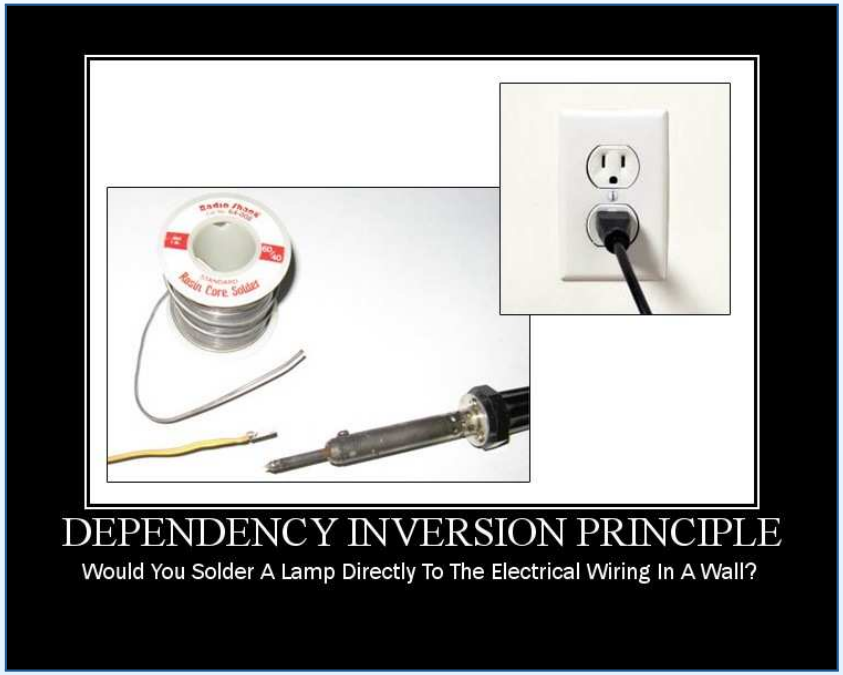
Principiul separării interfeţelor

- 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

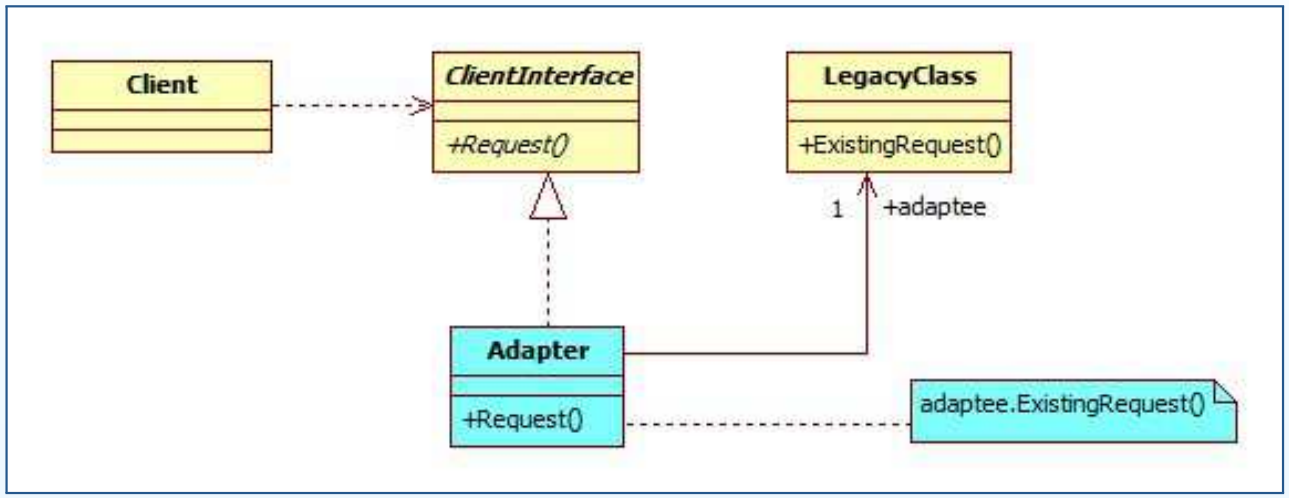
- Şabloane de proiectare adecvate acestor tipuri de schimbări
  - Furnizor nou / tehnologie nouă / implementare nouă
    - *Adapter*
    - *Bridge*
    - *Strategy*
  - Furnizor nou / tehnologie nouă
    - *Abstract Factory*
  - Funcţionalitate nouă
    - *Command*
  - O nouă complexitate a domeniului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



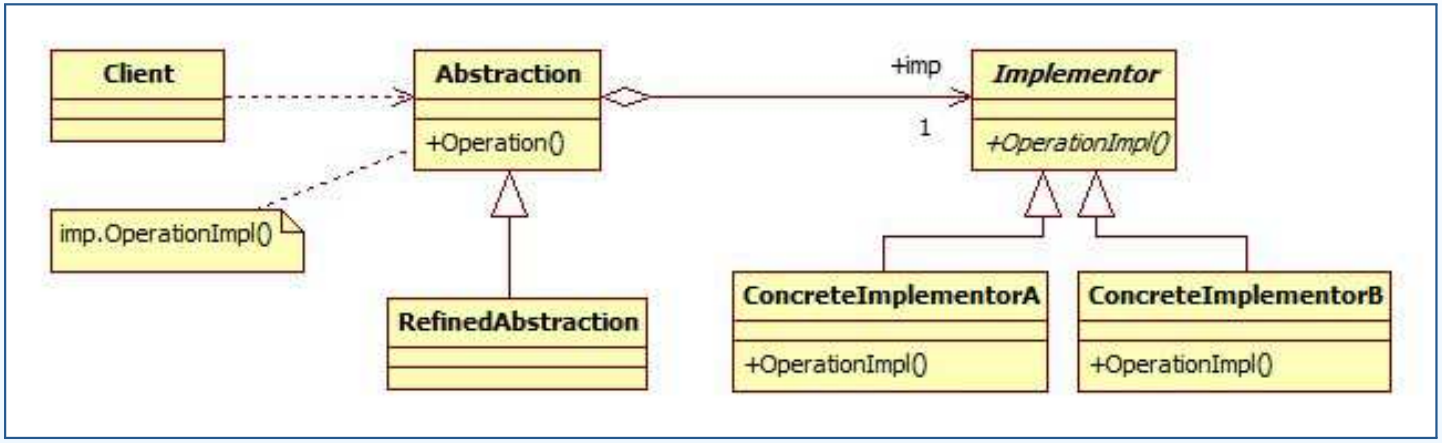
- *Consecinţe*:
  - Clasele *Client* şi *LegacyClass* pot colabora fără nici un fel de modificare la nivelul acestora
  - *Adapter* lucrează cu *LegacyClass* şi oricare dintre subclasele acesteia
  - Pentru fiecare specializare a *ClientInterface* trebuie scris un nou *Adapter*

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.

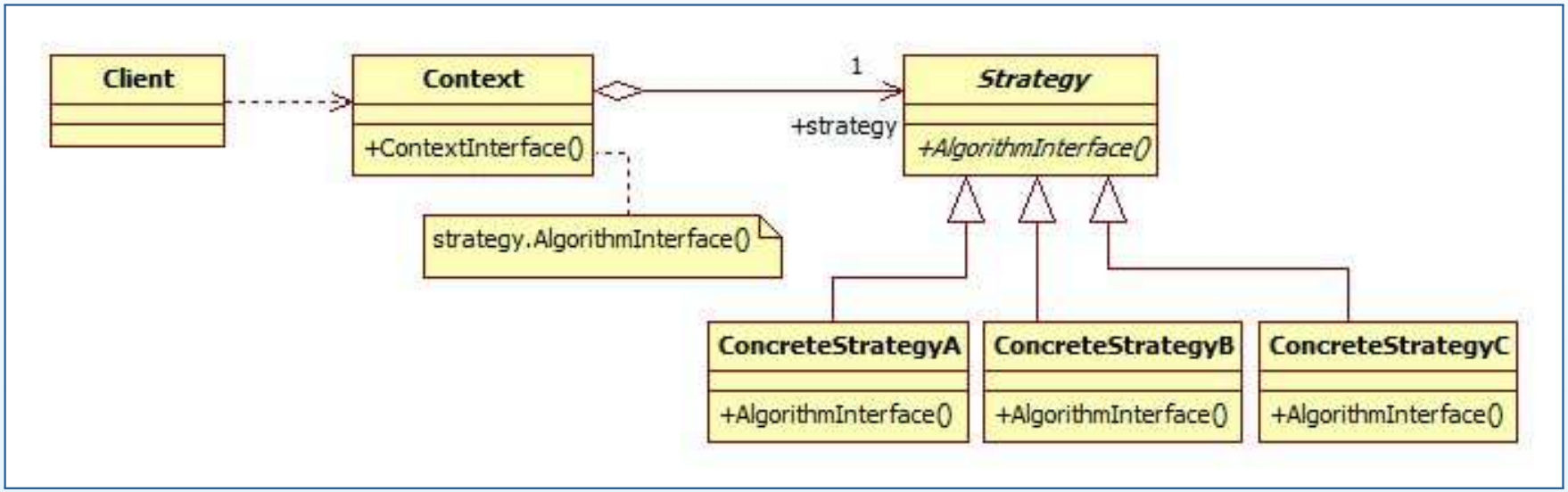
- *Consecinţe*:
  - Clientul este protejat de implementările abstracte şi concrete
  - Abstractizarea şi implementările pot fi rafinate independent

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*:

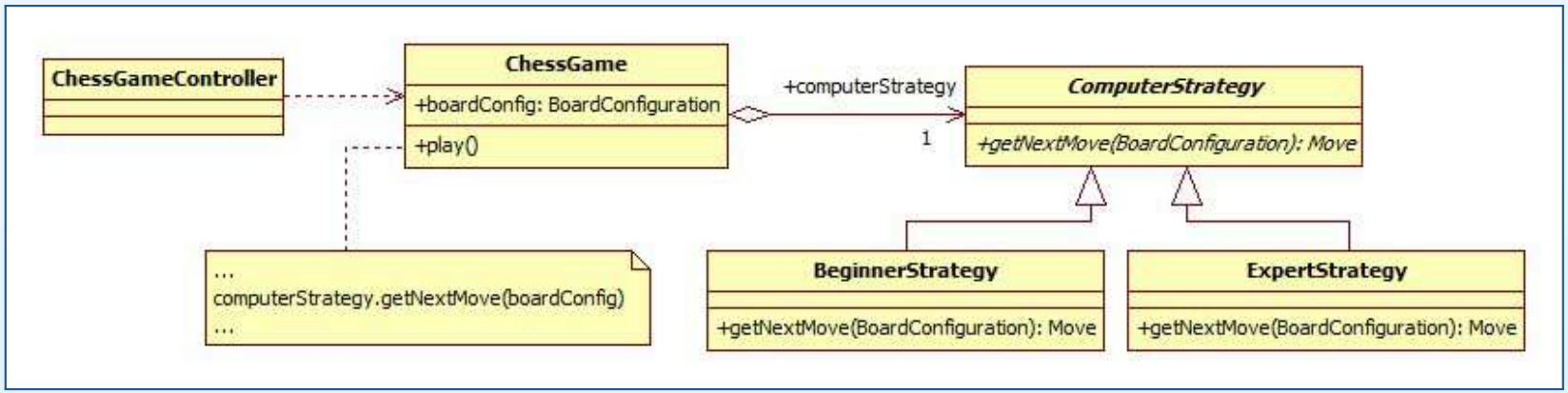


- *Soluţie*:
  - Fiecare dintre algoritmi este încapsulat/implementat de o clasă *ConcreteStrategy*. *Strategy* defineşte interfaţa comună a tuturor algoritmilor suportaţi.

Şablonul *Strategy* (cont.)

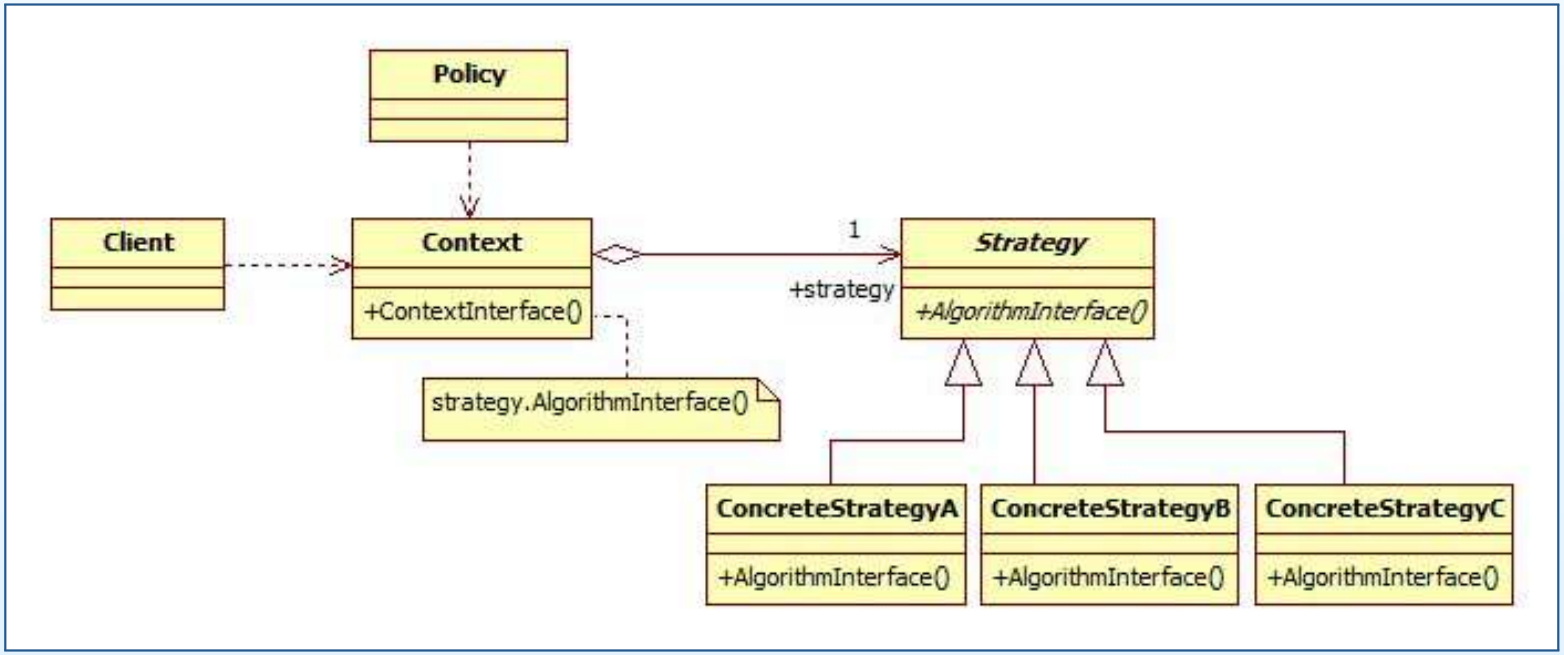
- Obiectele *Context* utilizează această interfaţă pentru a apela algoritmul definit de o clasă *ConcreteStrategy*. Contextul păstrează o referinţă către un obiect *Strategy* şi îi deleagă acestuia responsabilitatea execuţiei algoritmului, atunci când este cazul.
- Obiectul strategie este creat şi plasat 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



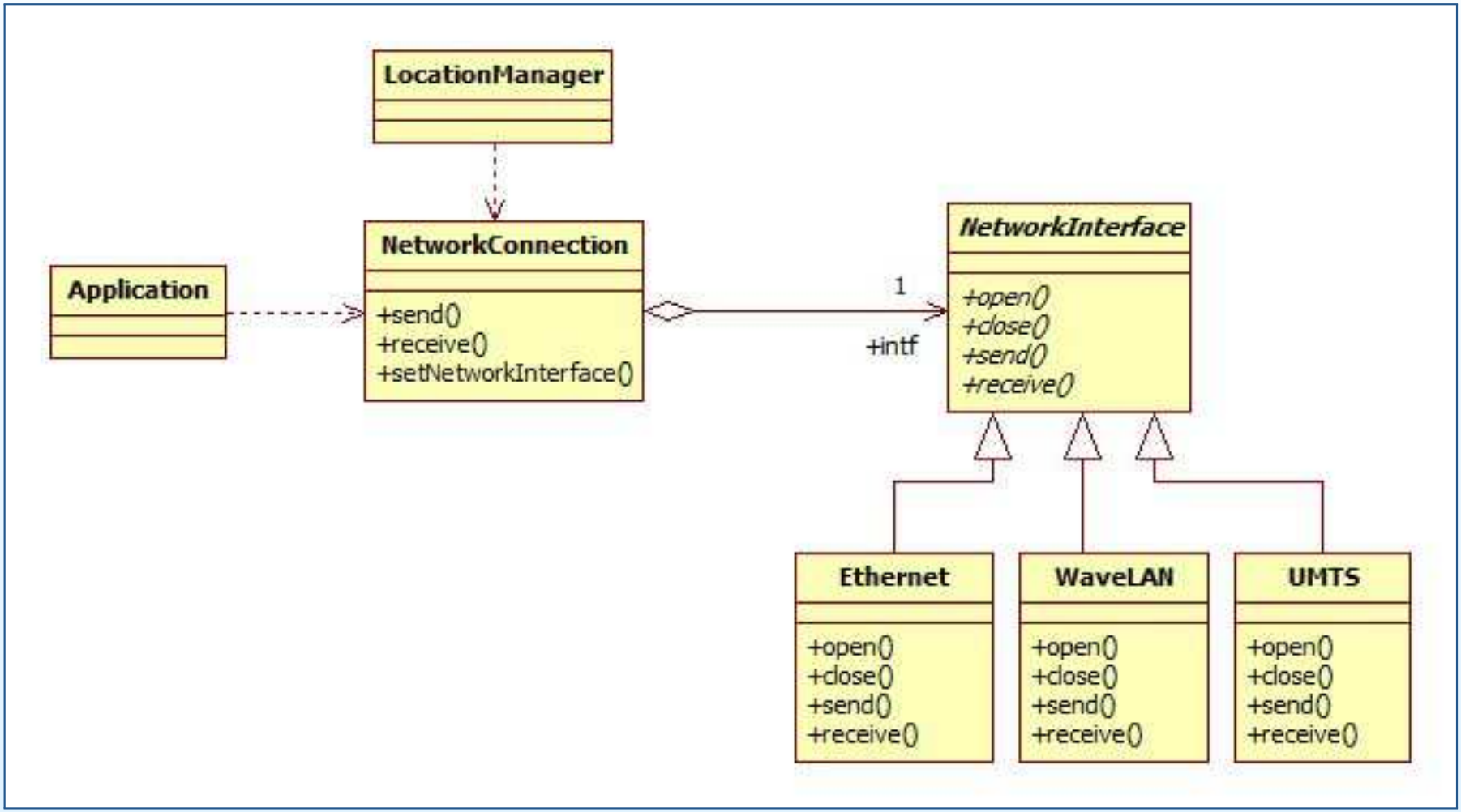
Şablonul *Strategy* (cont.)

- *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.)

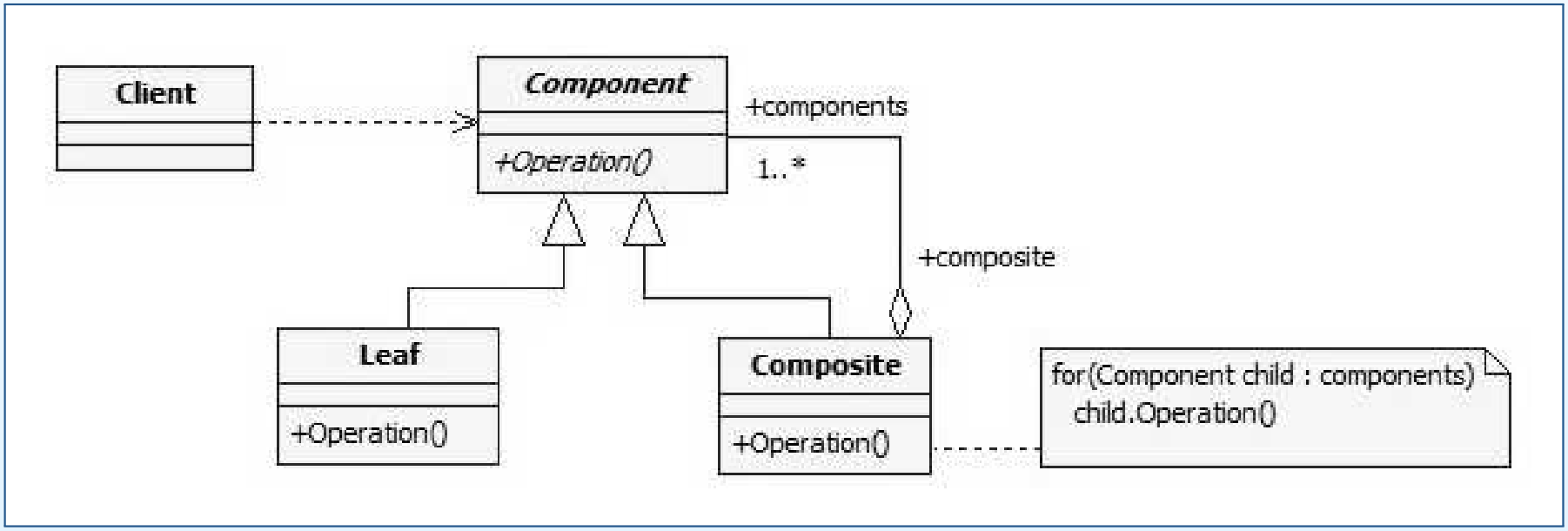
```
/** The NetworkConnection object represents a single abstract connection
 *  used by the Client. This is the Context object in Strategy pattern. */
public class NetworkConnection {
    private String destination;
    private NetworkInterface intf;
    private StringBuffer queue;

    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.)

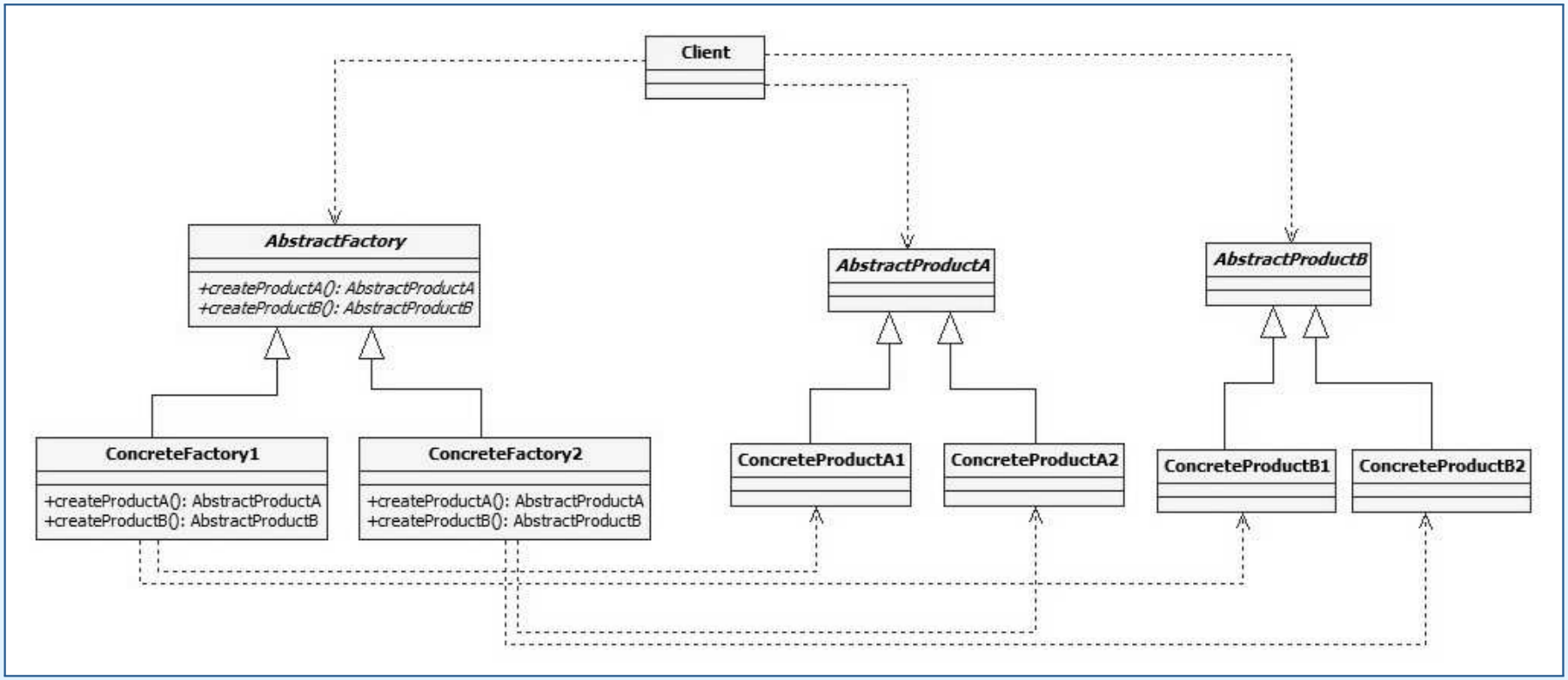
- *Soluţie*: Interfaţa *Component* specifică serviciile partajate de *Leaf* şi *Composite* (ex. *move*(*x*,*y*), pentru un obiect grafic). Clasa *Composite* agregă obiecte *Component* şi implementează aceste servicii iterând peste componentele conţinute şi delegându-le serviciul în cauză (ex. *move*(*x*,*y*) 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*:



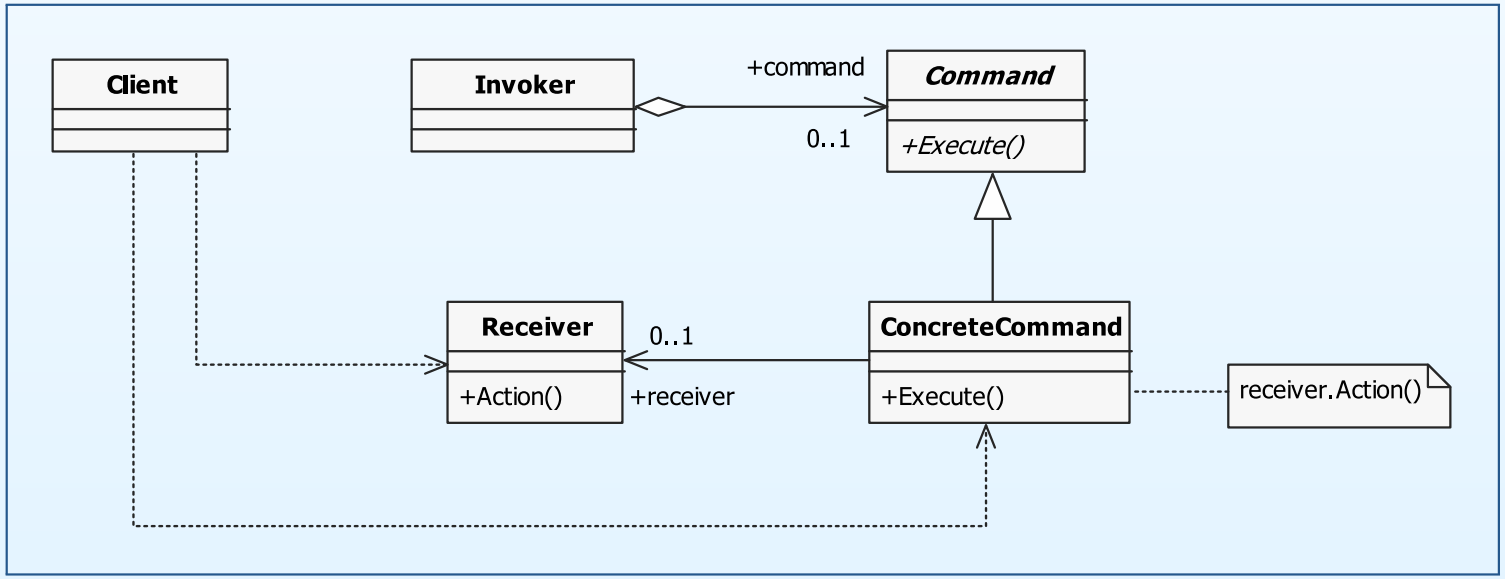
Şablonul *AbstractFactory* (cont.)

- *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.

Curs 9

Proiectarea obiectuală: Specificarea interfeţelor

Suport de curs bazat pe ***B. Bruegge and A.H. Dutoit***

"Object-Oriented Software Engineering using UML, Patterns, and Java"

Specificarea interfeţelor

- Scopul proiectării obiectuale este reprezentat de identificarea şi rafinarea obiectelor din domeniul soluţiei necesare realizării comportamentului subsistemelor identificate în etapa proiectării de sistem
- Produse existente până în momentul etapei de specificare a interfeţelor
  - *modelul obiectual de analiză*: entităţi din domeniul problemei (cu atribute, relaţii, unele operaţii) + obiecte boundary şi control inteligibile utilizatorului
  - *descompunerea sistemului*: subsisteme, servicii oferite, dependenţe între subsisteme, obiecte noi din domeniul soluţiei
  - *mapare hardware/software*: maşina virtuală = componente reutilizate pentru serviciile standard
  - *şabloane de proiectare reutilizate*, *componente de bibliotecă reutilizate pentru structuri de date şi servicii de bază*
- Subactivităţi în specificarea interfeţelor
  - Identificarea atributelor şi operaţiilor lipsă
  - Specificarea vizibilităţii şi signaturii operaţiilor
  - **Specificare pre/post-condiţiilor pentru operaţii şi a invarianţilor de tip**

Object Constraint Language (OCL)

- OCL [Warmer, 1999] - limbaj formal, utilizat pentru definirea de expresii pe modelele UML
  - introdus iniţial ca şi limbaj de modelare la IBM, astăzi standard OMG [OMG, 2014]
- Caracteristici OCL
  - *Limbaj complementar* (UML-ului)
    - OCL nu e un limbaj de sine stătător; a apărut din necesitatea de a acoperi problemele de expresivitate ale UML, a cărui natură diagramatică nu permite formularea multora dintre constrângerile caracteristice sistemelor nontriviale
    - Pentru dezvoltatori, specificaţiile OCL practice sunt doar cele formulate în contextul tipurilor de date utilizator introduse în modelul UML
  - *Limbaj declarativ* (limbaj de specificare pur, fără efecte secundare)
    - Evaluarea expresiilor OCL nu modifică starea modelului UML asociat
  - *Limbaj puternic tipizat*
    - Fiecare (sub)expresie OCL are un tip şi face obiectul verificărilor privind conformanţa tipurilor

Object Constraint Language (OCL) (cont.)

- *Limbaj bazat pe logica de ordinul întâi*
- *Limbaj care suportă principalele caracteristici OOP*
  - Specificaţiile OCL sunt moştenite în descendenţi, unde pot fi supradefinite
  - Redefinirea constrângerilor se conformează regulilor DbC
  - Limbajul suporta up/down-casting şi verificări de tip

- Utilitate
  - *navigarea modelului*
    - interogarea informaţiei din model, prin navigări repetate ale asocierilor, folosind nume de roluri
  - *specificarea aserţiunilor*
    - definirea explicită a pre/post-condiţiilor şi invarianţilor, conform principiilor DbC
  - *specificare comportamentală*
    - specificarea comportamentului observatorilor (operaţiilor de interogare) din model, specificarea regulior de derivare pentru atributele/referinţele derivate, definirea de noi atribute sau operaţii
  - *specificarea gărzilor*, *specificarea invarianţilor de tip pentru stereotipuri*

Sistemul de tipuri OCL

- OCL fiind complementar UML-ului, orice clasificator dintr-un model UML este un tip OCL valid in cadrul oricărei expresii ataşate modelului în cauză
- Biblioteca standard OCL - tipuri predefinite, independente de model
  - Tipuri primitive: `Integer`, `UnlimitedNatural`, `Real`, `Boolean`, `String`
  - Tipuri specifice OCL: `OclAny`, `OclVoid`, `OclInvalid`, `OclMessage`
  - Tipuri colecţie: `Collection`, `Set`, `OrderedSet`, `Sequence`, `Bag`
  - `Enumeration`, `TupleType`
- *Tipuri specifice OCL*
  - Tipul `OclAny`
    - Supertipul tuturor tipurilor OCL (=> în particular, fiecare clasă din modelul UML moşteneşte toate operaţiile definite în `OclAny`)
    - Operaţii
      `=(object2:OclAny):Boolean` - egalitatea a două obiecte
      `<>(object2:OclAny):Boolean` - egalitatea a două obiecte
      `oclIsTypeOf(type:Classifier):Boolean` - conformanţa tipurilor

Sistemul de tipuri OCL (cont.)

- Operaţii
  `oclIsKindOf(type:Classifier):Boolean` - conformanţa tipurilor
  `oclType():Classifier` - inferă tipul
  `oclAsType(type:Classifier):T` - conversie
  `oclIsNew():Boolean` - utilizat în postcondiţia unei operaţii, verifică dacă obiectul a fost creat în timpul execuţiei operaţiei respective
  `oclIsUndefined():Boolean` - verifică dacă obiectul există/e definit
  `oclIsValid():Boolean` - verifică dacă obiectul este valid

- Tipul `OclVoid`
  - Tip care se conformează tuturor tipurilor OCL, mai puţin `oclInvalid`
  - Denotă absenţa unei valori (sau o valoare necunoscută la momentul respectiv), singura valoare a tipului e literalul `null`

- Tipul `OclInvalid`
  - Tip care se conformează tuturor tipurilor OCL, inclusiv `OclVoid`
  - Singura valoare este `invalid`, ce poate rezulta din excepţii privind împărţirea la zero, accesarea unei valori de pe un index nepermis, etc.

Sistemul de tipuri OCL (cont.)

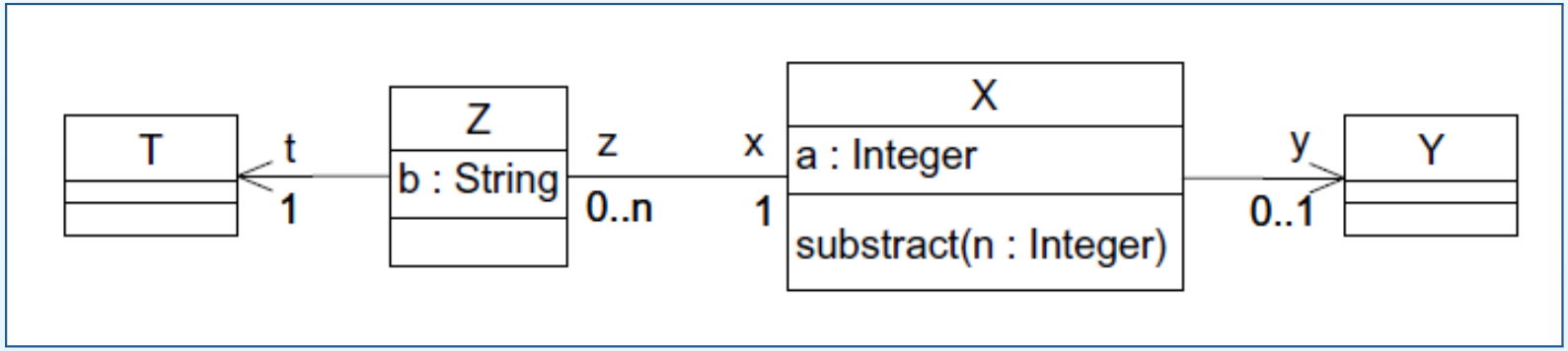
- Tpurile colecţie (tipuri template)
  - Operaţiile pe colecţii sunt apelate folosind notaţia $\rightarrow$
  - Modalităţi de a obţine o colecţie: prin navigare, ca rezultat al unei operaţii pe colecţii, folosind specificaţii cu literali (`Set{}`, `Bag{1,2,1}`, `Sequence{1..10}`)
  - Tipul `Collection`
    - Supertipul abstract al celorlalte tipuri colecţie din biblioteca standard OCL (Set, OrderedSet, Bag, Sequence)
    - Defineşte operaţii cu semantică comună tuturor subtipurilor
    - Unele operaţii sunt redefinite în subtipuri, având o postcondiţie mai puternică sau o valoare de retur mai specializată
    - Operaţii uzuale
      `size():Integer`, `isEmpty():Boolean`, `notEmpty():Boolean`
      `count(object:T):Integer`
      `includes(object:T):Boolean`, `excludes(object:T):Boolean`
      `includesAll(c2:Collection(T)):Boolean`
      `excludesAll(c2:Collection(T)):Boolean`
      `asSet():Set(T)`, `asOrderedSet():OrderedSet(T)`
      `asBag():Bag(T)`, `asSequence():Sequence(T)`

Sistemul de tipuri OCL (cont.)

- Iteratori pe colecţii
  - **select**: `source->select(iterator | body)` - returnează subcolecţia colecţiei `source` pentru care `body` se evaluează la `true`
  - **reject**: `source->reject(iterator | body)` - returnează subcolecţia colecţiei `source` pentru care `body` se evaluează la `false`
  - **forAll**: `source->forAll(iterator | body)` - returnează `true` dacă pentru toate elementele din colecţia `source` `body` se evaluează la `true`
  - **exists**: `source->exists(iterator | body)` - returnează `true` dacă există cel puţin un element în colecţia `source` pentru care `body` se evaluează la `true`
  - **one**: `source->one(iterator | body)` - returnează `true` dacă există exact un element al colecţiei `source` pentru cate `body` se evaluează la `true`
  - **any**: `source->any(iterator | body)` - returnează un element arbitrar din colecţia `source` pentru care `body` se evaluează la `true`
  - **isUnique**: `source->isUnique(iterator | body)` - returnează `true` dacă evaluările lui `body` conduc la elemente distincte
  - **colect**: `source->collect(iterator | body)` - returnează colecţia rezultată prin aplicarea lui `body` pe fiecare element al colecţiei sursă

Proprietăţi şi navigare

- O expresie OCL este formulată în contextul unui anumit tip
  - În cadrul respectivei expresii, instanţa contextuală este referită de cuvântul cheie `self`
  - `self` poate fi omis, atunci când nu există risc de ambiguităţi
  - Pornind de la instanţa contextuală, se pot accesa oricare dintre atributele, operaţiile de tip interogare sau capetele opuse de asociere, în stilul orientat-obiect clasic (folosind notaţia ".")
- Ex.:



  - În contextul clasei `X`, `self.a` şi `self.y` sunt două expresii OCL având tipurile `Integer`, respectiv `Y` (prima accesează un atribut, a doua presupune o navigare a unei asocieri folosind numele de rol al capătului opus)

Proprietăţi şi navigare (cont.)

- Reguli de tipizare la navigare
  - Atunci când multiplicitatea capătului opus de asociere este cel mult 1, tipul expresiei rezultate prin navigare într-un singur pas este dat de clasificatorul de la capătul opus
  - Atunci când valoarea maximă a multiplicităţii capătului opus de asociere este cel puţin 1, tipul expresiei rezultate prin navigare într-un singur pas este `Set` sau `OrderedSet`, funcţie de prezenţa sau absenţa constrângeii `{ordered}` pe capătul opus
    - În contextul `X`, tipul expresiei `self.z` este `Set(Z)`
  - Atunci când navigarea presupune mai mulţi paşi, tipul expresiei rezultat este `Bag`
    - În contextul `X`, tipul expresiei `self.z.t` este `Bag(T)`
- În afară de accesarea proprietăţilor instanţei contextuale, este posibilă utilizarea operaţiei `allInstances` pe un anumit clasificator => mulţimea tuturor obiectelor existente, având acel clasificator ca şi tip
  - `X.allInstances()->size()` - numărul obiectelor curente de tip `X`

Design by Contract în OCL

- Constrângeri de tip `invariant`

```
context X
 inv invX1: self.a >= 0
```

  - Un invariant se formulează în contextul unui clasificator, ce dă tipul instanţei contextuale
  - Un invariant este introdus de cuvântul cheie `inv`, urmat de un identificator opţional şi de expresia OCL a invariantului

- Constrângeri de tip `precondition/postcondition`

```
context X::substract(n:Integer)
 pre substractPre:   self.a >= n
 post substractPost: self.a = self.a@pre - n
```

  - Clauza `context` menţionează signatura operaţiei aferente (`self` va fi o instanţă a tipului care deţine acea operaţie)
  - Într-o postcondiţie, notatia `@pre` referă valoare unui obiect/unei proprietăţi înainte de execuţia operaţiei în cauză

Structurarea specificaţiilor OCL

- Mecanismul `let`
  - Permite extragerea unei subexpresii OCL redundante într-o variabilă
  - Creşte inteligibilitatea constrângerii şi eficienţa evaluării acesteia (prin efectuarea calculului aferent o singură dată)

```
context X
 inv invX2: let allT:Bag(T) = self.z.t in
            allT->size() = allT->asSet()->size()
```

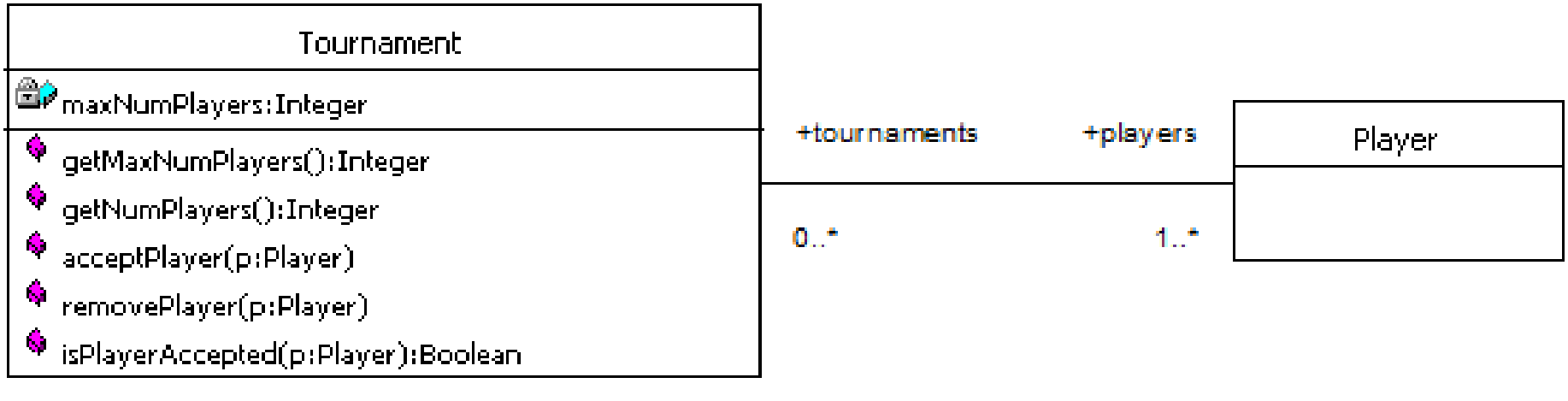
- Constrângeri de tip `definition`
  - Permit reutilizarea unei expresii OCL la nivelul mai multor constrângeri
  - Introduse prin cuvântul cheie `def`, permit definirea unor atribute/operaţii auxiliare la nivelul unui clasificator

```
context X
 def: hasY:Boolean = not self.y.oclIsUndefined()
 def: hasZWithBValue(value:String):Boolean =
       self.z->exists(zz | zz.b = value)
```

Iteratori - variante de sintaxă

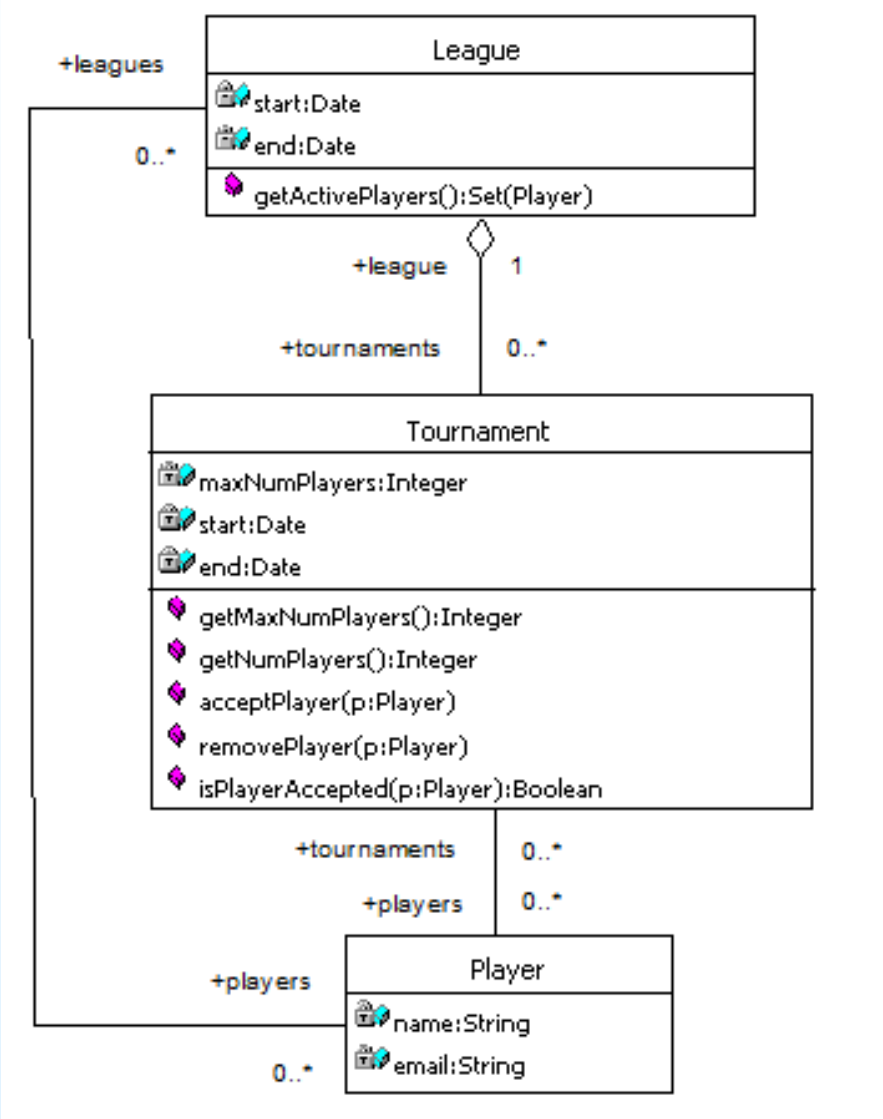
- `select` (analog `reject`, `forAll`, `exists`)
  - `collection->select(v:Type | boolean-expression-with-v)`
  - `collection->select(v | boolean-expression-with-v)`
  - `collection->select(boolean-expression)`
- `collect`
  - `collection->collect(v:Type | expression-with-v)`
  - `collection->collect(v | expression-with-v)`
  - `collection->collect(expression)`
- `iterate`
  - `collection->iterate(elem:Type; acc:Type = <expression> | expression-with-elem-and-acc)`
  - Cel mai generic iterator, ceilalţi pot fi exprimaţi folosindu-l pe `iterate`
  - Ex.: `collection->collect(x:T | x.property)` is equivalent to `collection->iterate(x:T; acc:Bag(T2) = Bag{} | acc->including(x.property))`

Exemple OCL

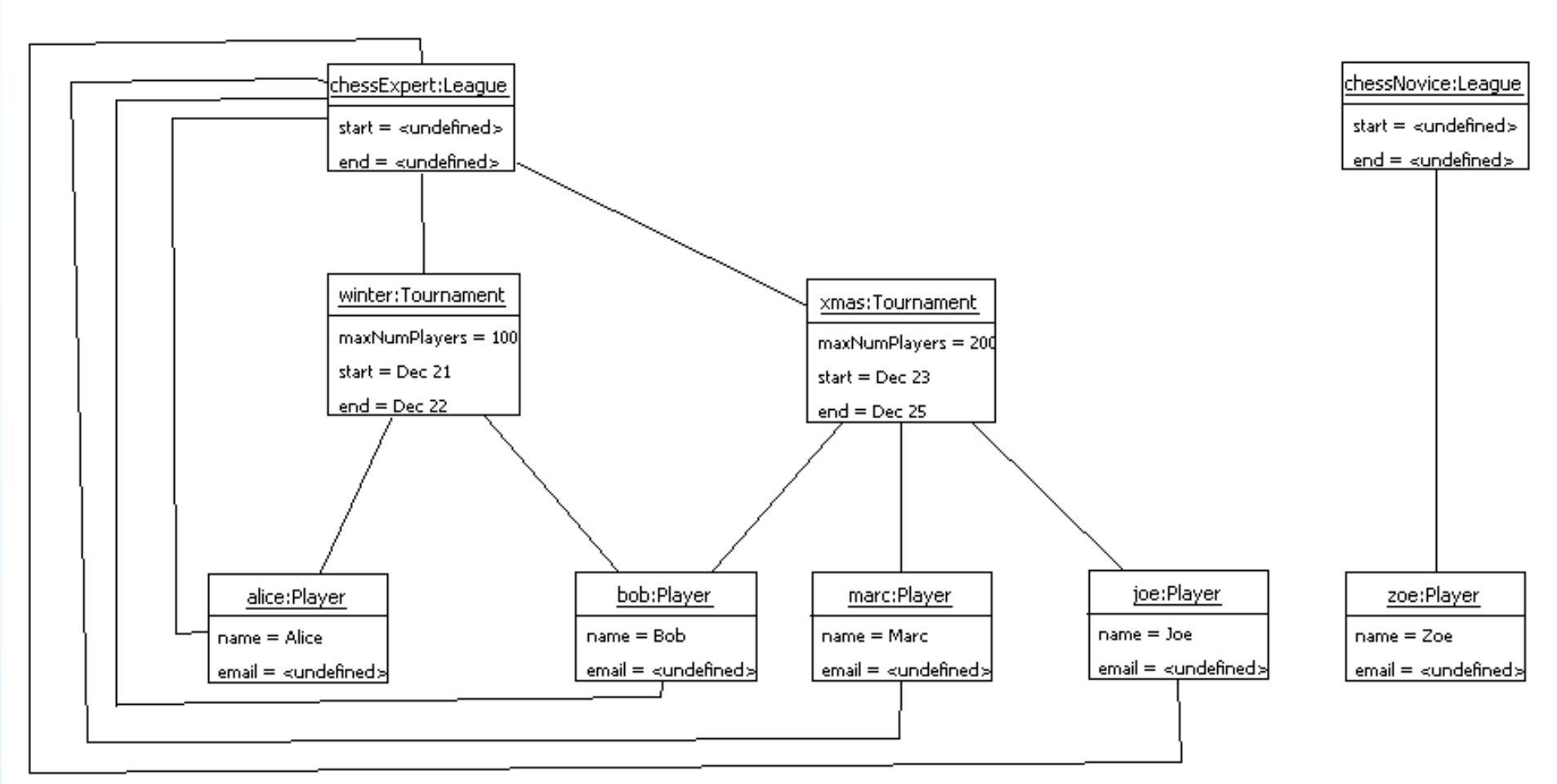



```
context Tournament
    inv maxNumPlayersPositive:
        self.getMaxNumPlayers() > 0

context Tournament::acceptPlayer(p:Player)
    pre:  self.getNumPlayers() < self.getMaxNumPlayers() and
          not self.isPlayerAccepted(p)
    post: self.isPlayerAccepted(p) and
          self.getNumPlayers() = self@pre.getNumPlayers() + 1
```

Exemple OCL (cont.)



Exemple OCL (cont.)



Exemple OCL (cont.)

- Durata unui turneu trebuie să fie sub o săptămână

```
context Tournament
   inv maxDuration: self.end - self.start < 7
```

- Toţi jucătorii care participă la un turneu trebuie să fie înregistraţi în liga aferentă acestuia

```
context Tournament
     inv allPlayersRegisteredWithLeague:
         self.league.players->includesAll(self.players)
```

```
context Tournament::acceptPlayer(p:Player)
    pre playerIsInLeague: self.league.players->includes(p)
```

Exemple OCL (cont.)

- Jucătorii activi dintr-o ligă sunt cei care au participat la cel puţin un turneu al ligii

```
context League::getActivePlayers():Set(Player)
    post: result = self.tournaments.players->asSet()
```

- Toate turneele unei ligi au loc în intervalul de timp aferent ligii

```
context League
    inv: self.tournaments->forAll(t:Tournament |
        t.start.after(self.start) and t.end.before(self.end)
```

- În orice ligă există cel puţin un turneu planificat în prima zi a ligii

```
context League
    inv: self.tournaments->exists(t:Tournament |
        t.start = self.start)
```

Moştenirea contractelor

- Problema moştenirii contractelor
  - În limbajele polimorfice, o referinţă la un obiect al clasei de bază poate fi substituită de o referinţă la un obiect al unei clase derivate
  - Codul client, scris în termenii clasei de bază, poate folosi obiecte ale claselor derivate, fără a avea cunoştinţă de acest fapt
  - => Clientul se aşteaptă ca un contract formulat relativ la clasa de bază, să fie respectat şi de clasele derivate
- Reguli privind moştenirea contractelor (consecinţă a principiului Liskov al substituţiei)
  - *Precondiţii*: Unei metode dintr-o subclasă îi este permis să slăbească precondiţia metodei pe care o supradefineşte (o metodă care supradefineşte poate gestiona mai multe cazuri decât cea supradefinită)
  - *Postcondiţii*: O metodă care supradefineşte trebuie să asigure o postcondiţie cel puţin la fel de puternică precum cea supradefinită
  - *Invarianţi*: O subclasă trebuie să respecte toţi invarianţii superclaselor sale; poate, eventual, introduce invarianţi mai puternici decăt cei moşteniţi

Referinţe

- [OMG, 2014] Object Management Group, *Object Constraint Language - version 2.4*, February 2014.
- [Warmer, 1999] J. Warmer, A. Kleppe, *Object constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.

Design by Contract (DbC) [1,2]

- The Design by Contract (DbC) methodology has entered software development due to Bertrand Meyer, along with the Eiffel language
- It proposes a contractual approach to the development of object-oriented software components, based on the use of *assertions*
- The approach has been aimed at increasing the *reliability* of object-oriented software components - a critical requirement in the context of large-scale software reuse, as promoted by the object-oriented paradigm
  - *reliability* = *correctness* (software's ability to behave according to the specification) + *robustness* (the ability to properly handle situations outside of the specification)
- Expected to positively contribute to
  - the develoment of correct and robust OO systems
  - a deeper understanding of inheritance and related concepts (overriding, polymorphism, dynamic bynding), by means of the *subcontracting* concept
  - a systematic approach to *exception handling*

Software Contracts. Assertions

- A generic algorithm to solve a non-trivial task

```
Algorithm mainTask is:
  @subTask_1;
  @subTask_2;
  ...
  @subTask_n;
End-mainTask
```

- Each subtask may be either inlined or triggering the call of a subroutine
- Analogy: calling a subroutine to solve a subtask vs. a real-life situation with a person (client) requiring the services of a third-party (provider) to accomplish a task that he cannot / would not do personally
  - e.g. contracting the services of a fast courier to deliver a package to a particular destination in a foreign city

Software Contracts. Assertions

- Characteristics of human contracts involving two parts
  - stipulate the benefits and obligations of each part; a benefit for one part is an obligation for the other
  - may also reference general laws that should be obeyed by both parts
- *Same principles should apply to software development: each time a routine depends on the call of a subroutine to accomplish a subtask, there should be a contractual specification among the client (caller) and supplier (calee)*
- The clauses of software contracts are formalized by means of *assertions*
  - *assertion = expression involving a number of software entities, which state a property that these entities should fulfill at runtime*
  - *closest concept – predicate, implementation – boolean expressions*
  - *some apply to individual routines (pre/post-conditions), other to the class as a whole (invariants)*

Pre/post-condition assertions

- A <pre, post> pair for a method expresses the software contract that the method in question (provider of some service) publishes for its callers (clients of the service)
- Characteristics of software contracts
  - the precondition defines the situation when a call is legitimate - obligation for the client (caller) and benefit for the provider (method)
  - the postcondition states which properties should be fulfilled once the execution has ended - obligation for the provider (method) and benefit for the client (caller)
- The major contribution brought by DBC in the field of software reliability: precondition = benefit of the service provider
- *DBC non-redundancy principle: The body of a mehod should never check for a precondition* (as opposed to defensive programming)
- In Eiffel, assertions are part of the language, allowing for runtime monitoring
  - precondition violation = bug in the client, postcondition violation = bug in the supplier

Invariant assertions

- In addition to pre/post-conditions, that capture the behavior of individual methods, it is possible to express global properties of a class' instances, that should be preserved by all its public methods
- An *invariant* encloses all the semantic constraints and integrity rules applying to the class in question
- Lifecycle of an object

- An assertion *I* is a correct invariant for a class *C* if and only if the following conditions hold
  - each constructor of *C*, apply to arguments that fulfill its precondition, in a state in which the class attributes have default values, leads to a state in which *I is fulfilled*
  - each public method of *C*, applied to a set of arguments and to a state satisfying both *I* and the method precondition, leads to a state satisfying *I*

Correctness of a class

- Informally, a class is said to be correct with respect to its specification if and only if its implementation, as given by the method bodies, is consistent with its preconditions, postconditions and invariant
- **Definition:** The class *C* is said to be correct with respect to its assertions (pre/post-conditions and invariant) if and only if the following conditions hold:

  (1) [*default_C* and *pre_p(x_p)*] *body_p* [*post_p(x_p)* and *INV*]

  for each class constructor *p* and each set of valid arguments of *p – x_p* and

  (2) [*pre_r(x_r)* and *INV*] *body_r* [*post_r(x_r)* and *INV*]

  for each public class method r and each set of valid arguments of *r – x_r*, where

  default_C is an assertion stating that the attributes of *C* have default values for their type, *INV* is the invariant of *C*, *pre_m*, *post_m*, *body_m* are the precondition, postcondition and body of an arbitrary method *m* of *C*.

The purpose of using assertions

- Support in writting correct software, including the means to formally define correctness
  - The writing of explicit contracts comes as a prerequisite of their enforcement in software
- Support for a better software documentation
  - Essential when it comes to reusable assets, see the case of Ariane!
- Support for testing, debugging and quality assurance
  - Levels of runtime assertion monitoring:
    - 1.preconditions only
    - 2.preconditions and postconditions
    - 3.all assertions
  - While testing, enforce level 3, in production, there is a tradeoff between trust in the code, efficiency level desired and critical nature of the application
- Support for the development of fault tolerant systems

Defensive Programming [3]

- Analogy to *defensive driving*
  - *Defensive driving*: You can never be sure what the others might do, so take responsibility of protecting yourself, such that another driver's mistake won't hurt you!
  - *Defensive programming*: If a routine is passed bad data, it should not be hurt, even if the bad data is someone else's fault (humans, software).
- The core idea of defensive programming is guarding against unexpected errors
- Acknowledges that errors happen and invites programmers to write code accordingly
- Comprises a set of techniques that make errors easier to detect, easier to repair and less damaging in production code
- Should serve as a complement to defect-prevention techniques (iterative design, pseudocode first, design inspections, etc.)
- Protecting from invalid input involves
  - Checking all data received from the outside (from users, files, network, etc.)
    - numeric values should be between tolerances, strings – short enough to handle and obeying to their intended semantics
  - Checking all input parameters
  - Deciding on how to deal with bad data

References

[1] Meyer, B., *Object-Oriented Software Construction (2nd ed.)*, Prentice-Hall, 1997. (Chapter 11 – Design by Contract: building reliable software)

[2] Meyer, B., *Applying „Design by Contract”*, IEEE Computer 25(10):40-51, 1992.

[3] McConnel, S., *Code Complete (2nd ed.)*, Microsoft Press, 2004. (Chapter 8 – Defensive Programming)

Curs 10

Transformarea modelelor în cod

Suport de curs bazat pe ***B. Bruegge and A.H. Dutoit***

"Object-Oriented Software Engineering using UML, Patterns, and Java"

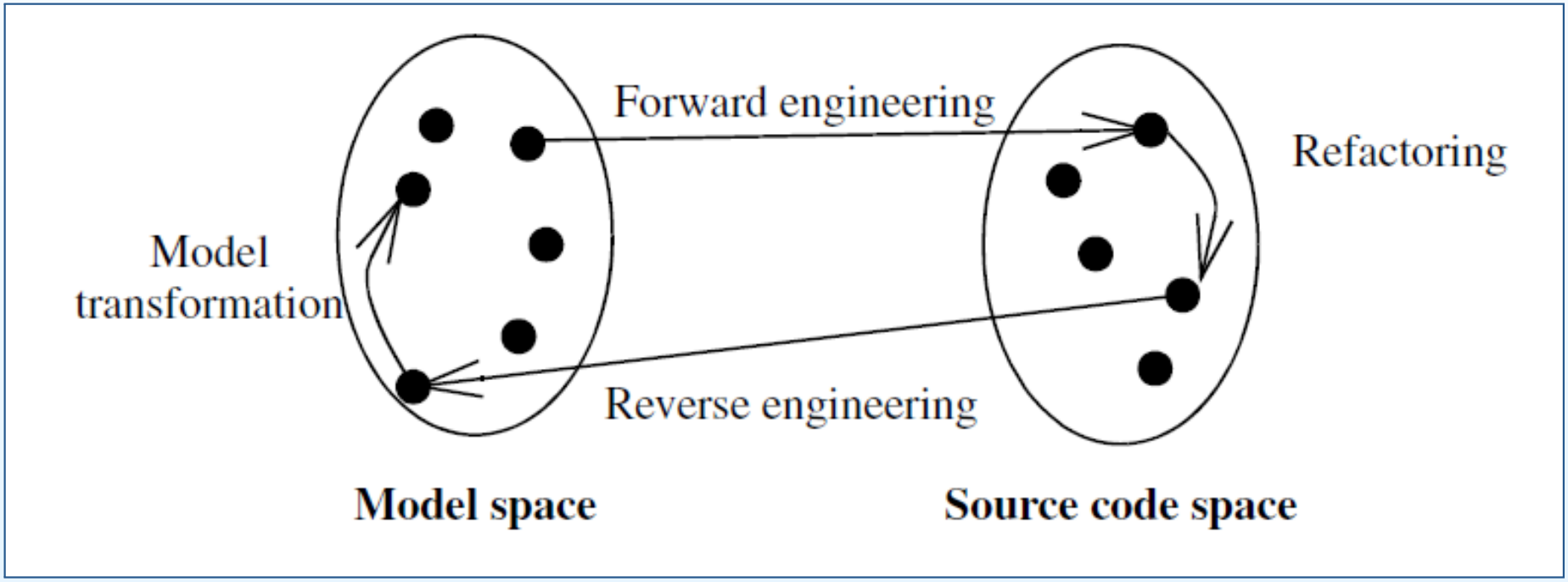
Modele şi transformări

- O *transformare* are drept scop îmbunătăţirea unei caracteristici a unui model (ex.: modularitatea), cu păstrarea celorlalte proprietăţi ale acestuia (ex.: funcţionalitatea)
  - O transformare este, de obicei, localizată, afectează un număr relativ mic de clase/atribute/operaţii şi se execută într-o succesiune de paşi mărunţi
- Astfel de transformări caracterizează preponderent activităţile legate de proiectarea obiectuală şi implementarea sistemului
  - *Optimizare* - îndeplinirea cerinţelor legate de performanţa sistemului, prin
    - reducerea multiplicităţii asocierilor, pentru a creşte viteza interogărilor
    - adăugarea unor asocieri redundante, pentru eficienţă
    - introducerea unor atribute derivate, pentru a îmbunătăţi timpul de acces la obiecte
  - *Reprezentarea asocierilor* - implementarea asocierilor în cod folosind (colecţii de) referinţe
  - *Reprezentarea contractelor* - descrierea comportamentului sistemului în cazul violării contractelor, folosind excepţii
  - *Reprezentarea entităţilor persistente* - maparea claselor la nivelul depozitelor de date (baze de date, fisiere text, etc.)

Tipuri de transformări

- *Transformări la nivelul modelului* (eng. *model transformations*)
  - Operează pe un model, au ca şi rezultat un model
  - Ex.: transformarea unui atribut (atribut *adresă*, reprezentată ca şi string) într-o clasă (clasa *Adresă*, cu atribute *stradă*, *număr*, *oraş*, *cod poştal* etc.)
- *Refactorizări* (eng. *refactorings*)
  - Operează pe cod sursă, au ca şi rezultat cod sursă
  - Similar transformărilor la nivelul modelului, îmbunătăţesc un aspect al sistemului, fără a-i afecta funcţionalitatea
- *Inginerie directă* (eng. *forward engineering*)
  - Produce un fragment de cod aferent unui model obiectual
  - Multe dintre conceptele de modelare (ex.: atribute, asocieri, signaturi de operaţii) pot fi transformate automat în cod sursă; corpul metodelor, precum şi metodele adiţionale (private) sunt inserate manual de către dezvoltatori
- *Inginerie inversă* (eng. *reverse engineering*)
  - Produce un model, pe baza unui fragment de cod sursă
  - Utilă atunci când modelul de proiectare nu (mai) există sau atunci când modelul şi codul au evoluat desincronizat

Tipuri de transformări (cont.)

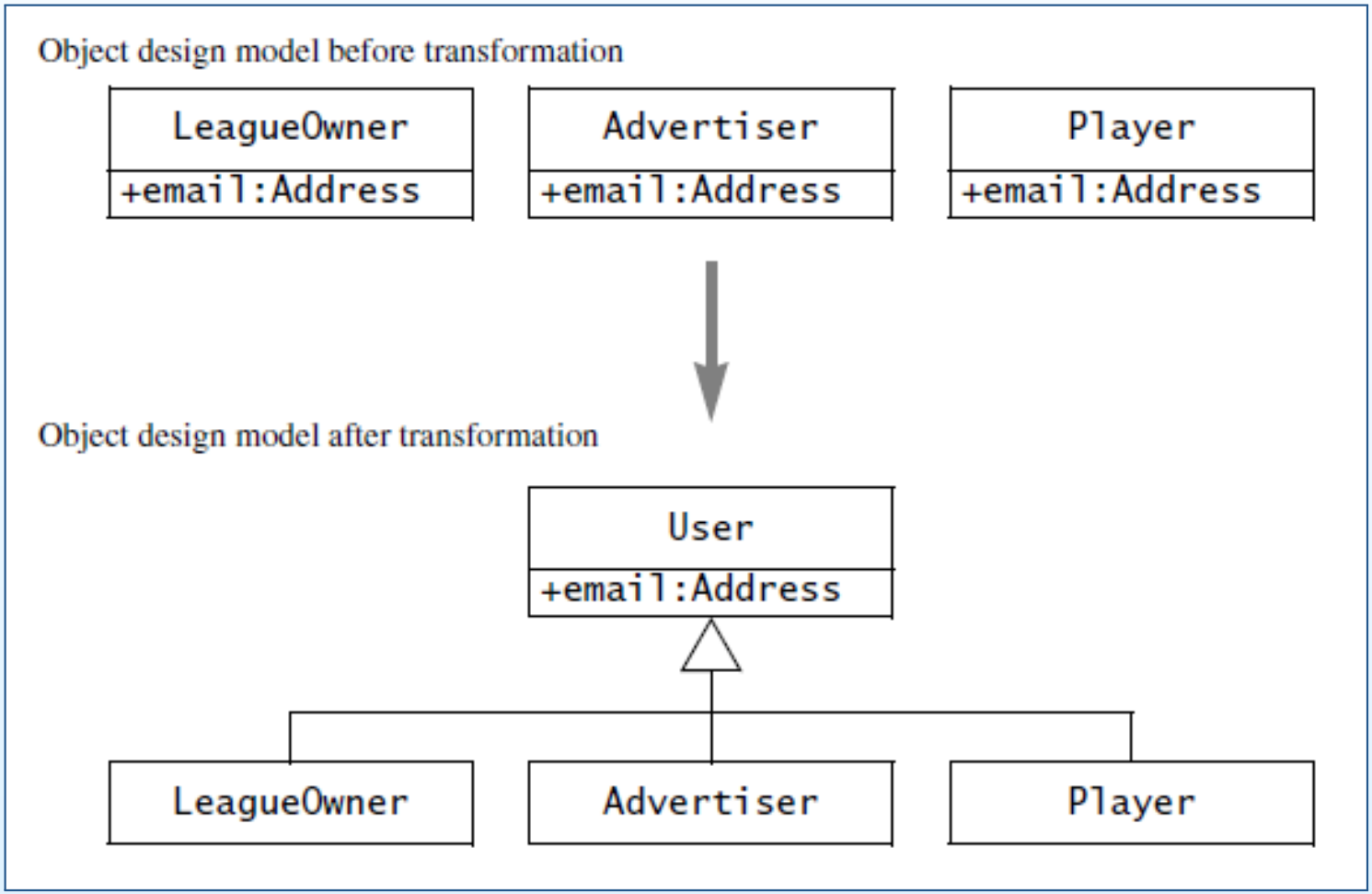


Transformări la nivelul modelului

- O astfel de transformare este aplicată unui model obiectual şi rezultă într-un nou model obiectual
- Obiectivul este simplificarea, detalierea sau optimizarea modelului iniţial, în conformitate cu cerinţele din specificaţie
- O transformare la nivelul modelului poate să adauge, să elimine sau să redenumească clase, operaţii, asocieri sau atribute
- Întreg procesul de dezvoltare poate fi considerat ca şi o succesiune de transformări de modele, începând cu modelul de analiză şi terminând cu cel obiectual de proiectare, fiecare astfel de transformare adăugând detalii care ţin de domeniul soluţiei
- Deşi aplicarea unei astfel de transformări poate fi, de cele mai multe ori, automatizată, identificarea tipului de transformare de aplicat, precum şi a claselor concrete implicate necesită raţionament şi experienţă

Transformări la nivelul modelului (cont.)

- *Ex.10.1*: utilizarea unei transformări pentru introducerea unei ierarhii de clase şi eliminarea redundanţei din modelul obiectual de analiză



Refactorizări

- O *refactorizare* reprezintă o transformare a codului sursă, care creşte inteligibilitatea sau modificabilitatea acestuia, fără a-i schimba comportamentul [Fowler, 2000]
  - O refactorizare are drept scop îmbunătăţirea design-ului unui sistem funcţional, focusându-se pe o anumită metodă sau pe un anumit câmp al unei clase
  - Pentru a asigura păstrarea neschimbată a comportamentului sistemului, o refactorizare se realizează incremental, paşii de refactorizare fiind intercalaţi cu teste
- *Ex.10.2*: Transformarea de model din Ex.10.1 corespunde unei serii de 3 refactorizări
  1. Refactorizarea *Pull Up Field*
     - Transferă câmpul *email* din subclase în superclasa *User*
  2. Refactorizarea *Pull Up Constructor Body*
     - Transferă codul de iniţializare din subclase în superclasă
  3. Refactorizarea *Pull Up Method*
     - Transferă metodele care utilizează câmpul *email* din subclase în superclasă

Paşii refactorizării *Pull Up Field*

1. Inspectează clasele *Player*, *LeagueOwner* şi *Advertiser*, pentru a certifica echivalenţa semantică a atributelor de tip *e-mail*. Redenumeşte atributele echivalente la *email*, dacă este necesar
2. Creează clasa publică *User*
3. Asignează clasa *User* ca şi superclasă pentru *Player*, *LeagueOwner* şi *Advertiser*
4. Adaugă câmpul protected *email* clasei *User*
5. Şterge câmpul *email* din clasele *Player*, *LeagueOwner* şi *Advertiser*
6. Compilează şi testează

Before refactoring

```
public class Player {
    private String email;
    //...
}
public class LeagueOwner {
    private String eMail;
    //...
}
public class Advertiser {
    private String email_address;
    //...
}
```

After refactoring

```
public class User {
    protected String email;
}
public class Player extends User {
    //...
}
public class LeagueOwner extends User
{
    //...
}
public class Advertiser extends User {
    //...
}
```

Paşii refactorizării *Pull Up Constructor Body*

1. Adaugă clasei *User* constructorul *User(String email)*
2. În constructor, asignează câmpului *email* valoarea transmisă ca şi parametru
3. Înlocuieşte corpul constructorului clasei *Player* cu apelul *super(email)*
4. Compilează şi testează
5. Repetă paşii 1-4 pentru *LeagueOwner* şi *Advertiser*

Before refactoring

```
public class User {
    private String email;
}


public class Player extends User {
    public Player(String email) {
        this.email = email;
        //...
    }
}
public class LeagueOwner extends User
{
    public LeagueOwner(String email) {
        this.email = email;
        //...
    }
}
public class Advertiser extends User {
    public Advertiser(String email) {
        this.email = email;
    //...
    }
}
```

After refactoring

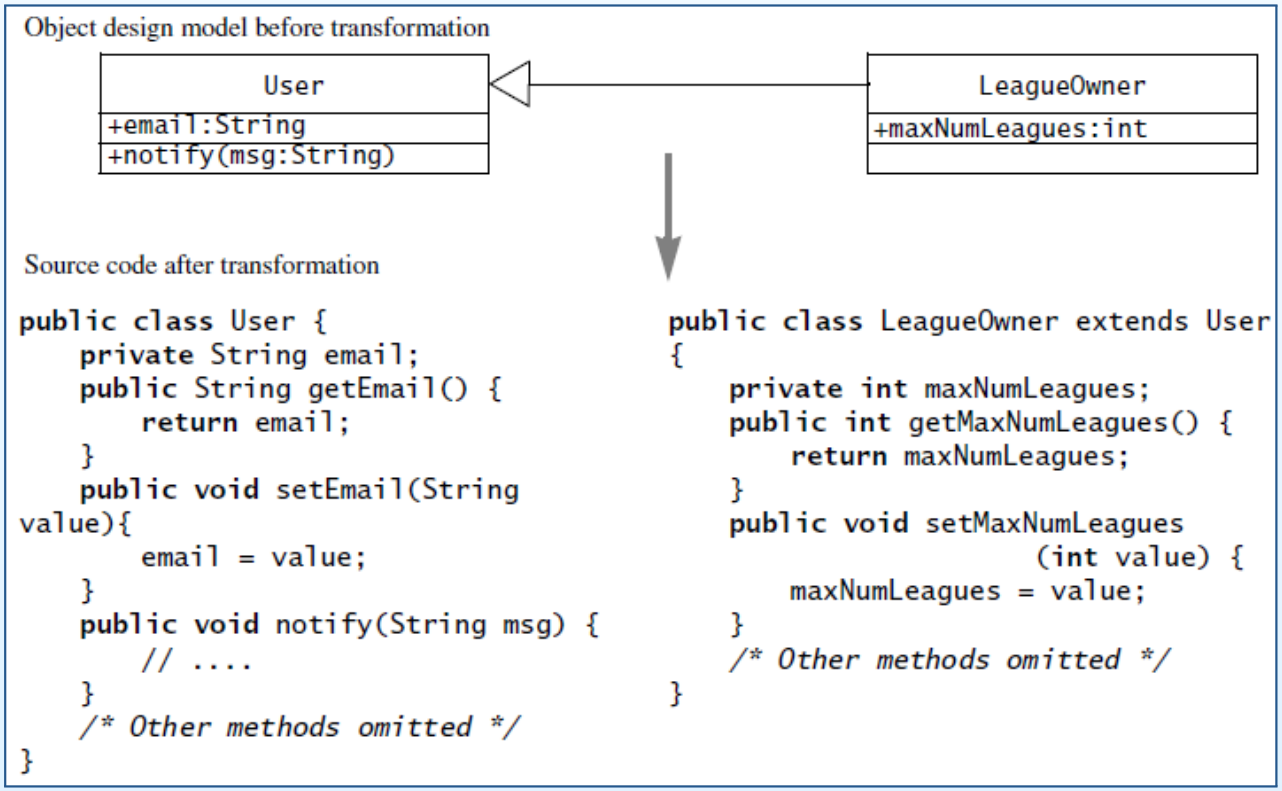
```
public class User {
    public User(String email) {
        this.email = email;
    }
}
public class Player extends User {
    public Player(String email) {
        super(email);
    //...
    }
}
public class LeagueOwner extends User
{
    public LeagueOwner(String email) {
        super(email);
    //...
    }
}
public class Advertiser extends User {
    public Advertiser(String email) {
        super(email);
    //...
    }
}
```

Paşii refactorizării *Pull Up Method*

1. Examinează metodele din *Player* care utilizează câmpul *email*. Presupunem că *Player.notify()* utilizează acest câmp, însă nu foloseşte nici un alt câmp şi nici o altă operaţie specifice lui *Player*
2. Copiază metoda *notify()* în clasa *User* şi recompilează
3. Şterge metoda *Player.notify()*
4. Compilează şi testează
5. Repetă paşii 1-4 pentru *LeagueOwner* şi *Advertiser*

Inginerie directă

- *Ingineria directă* se aplică unei mulţimi de elemente din model şi rezultă într-o mulţime de instrucţiuni într-un limbaj de programare (cod sursă)
  - Scopul ingineriei directe este acela de a întreţine o corespondenţă între modelul obiectual de proiectare şi cod şi de a reduce numărul de erori introduse la implementare (diminuând astfel efortul de implementare)



Inginerie inversă

- *Ingineria inversă* se aplică unei mulţimi de elemente din codul sursă, rezultând într-o mulţime de elemente de model
  - Scopul ingineriei inverse este acela de a recrea modelul aferent unui sistem, ca urmare a inexistenţei/pierderii sale sau a lipsei de sincronizare a acestuia cu codul sursă
  - Este transformarea opusă ingineriei directe (creează o clasă UML pentru fiecare declaraţie de clasă din codul sursă, adaugă un atribut pentru fiecare câmp al clasei, o operaţie pentru fiecare metodă)
  - Dat fiind că, prin inginerie directă, se pierde informaţie din model (ex. asocierile sunt convertite în referinţe sau colecţii de referinţe), ingineria inversă nu va produce, de regulă, acelaşi model
  - Majoritatea instrumentelor CASE existente cu suport integrat pentru ingineria inversă oferă cel mult o aproximare care permite dezvoltatorului reconstituirea modelului iniţial

Principii de transformare

- 1. Fiecare transformare trebuie să vizeze optimizări din perspectiva unui singur criteriu
  - Ex.: o aceeaşi transformare nu poate avea drept scop diminuarea timpului de răspuns al sistemului şi creşterea inteligibilităţii codului
  - Încercarea de a adresa mai multe criterii printr-o aceeaşi transformare creşte complexitatea transformării şi oferă condiţii pentru introducerea unor erori
- 2. Fiecare transformare trebuie să fie locală
  - O transformare trebuie să afecteze doar un număr mic de metode/clase la un moment dat
  - O modificare la nivelul implementării unei metode nu va afecta clienţii acesteia
  - Dacă transformarea vizează o interfaţă, clienţii trebuie modificaţi pe rând
- 3. Fiecare transformare trebuie aplicată izolat de alte schimbări
  - Ex.: Adăugarea unei noi funcţionalităţi şi optimizarea codului existent nu se vor opera simultan

Principii de transformare (cont.)

- 4. Fiecare transformare trebuie urmată de validări aferente
  - O transformare care operează doar asupra modelului trebuie urmată de modificarea diagramelor de interacţiune afectate de schimbarea de model efectuată şi de revizuirea cazurilor de utilizare aferente, pentru a certifica oferirea funcţionalităţii dorite
  - O refactorizare trebuie urmată de execuţia cazurilor de test aferente claselor afectate de schimbările efectuate
  - Introducerea unor noi funcţionalităţi trebuie urmată de proiectarea unor cazuri de test aferente

Optimizarea modelului obiectual de proiectare

- Are drept scop îndeplinirea criteriilor de performanţă ale sistemului (legate de timp de răspuns/execuţie sau spaţiu de memorare)
- Tipuri comune de optimizări
  - Optimizarea căilor de acces
  - Transformarea unor clase în atribute
  - Amânarea operaţiilor costisitoare
  - Memorarea (eng. *caching*) rezultatelor operaţiilor costisitoare
- Trebuie menţinut un echilibru între eficienţă şi claritate, întrucât transformările care vizează eficientizarea codului, au, de obicei, efecte negative asupra inteligibilităţii sistemului

Optimizarea căilor de acces

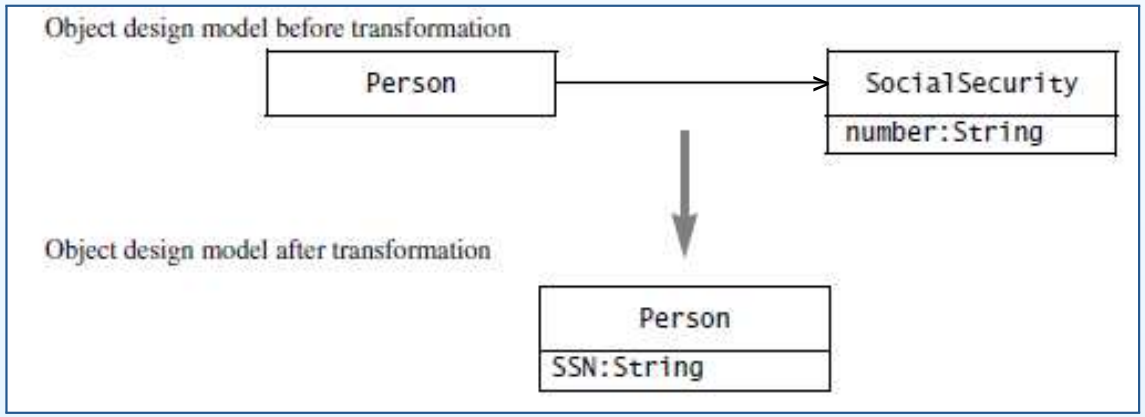
- Surse comune de ineficienţă la nivelul unui model obiectual
  - Traversarea repetată a unui număr mare de asocieri
  - Traversarea asocierilor cu multiplicitate *many*
  - Plasarea eronată a unor atribute
- Rezolvarea acestor probleme conduce la un model cu asocieri redundante intenţionate, un număr mai mic de relaţii cu multiplicităţi *many* şi un număr mai mic de clase
- *Traversarea repetată a unui număr mare de asocieri*
  - Operaţiile care trebuie executate frecvent şi presupun traversarea unui număr mare de asocieri introduc probleme de eficienţă
  - Identificarea acestora se realizează urmărind diagramele de interacţiune aferente cazurilor de utilizare
  - Soluţia: introducerea unor asocieri directe, redundante, între entităţile interogate şi cele care interoghează
  - De cele mai multe ori, aceste transformări se aplică doar în urma testării sistemului, după confirmarea, la execuţie, a problemelor de eficienţă anticipate

Optimizarea căilor de acces (cont.)

- *Traversarea asocierilor cu multiplicitate many*
  - Soluţii: calificarea asocierilor în scopul reducerii multiplicităţii; ordonarea sau indexarea obiectelor de la capătul aferent multiplicităţii *many*
- *Plasarea greşită a unor atribute*
  - Apare ca şi rezultat al modelării excesive/exagerate în etapa de analiză
  - Soluţie: atribute ale unor clase fără comportament relevant (doar metode get/set) pot fi relocate în clasa apelantă
  - Astfel de relocări pot conduce la eliminare din model a unor clase
- **ToDo :)** Imaginaţi-vă câte o situaţie de fiecare dintre cele trei tipuri enumerate şi soluţia de modelare aferentă (model iniţial vs. model după transformare). Pentru obţinerea unui bonus la curs, trimiteţi răspunsul pe adresa `vladi@cs.ubbcluj.ro` până la finalul zilei in care a fost postat materialul de curs pe pagină.

Transformarea unor clase în atribute

- După restructurări/optimizări repetate ale modelului obiectual, unele clase vor rămâne cu un număr mic de atribute/operaţii
- Astfel de clase, atunci când sunt asociate cu o singură altă clasă, pot fi contopite cu aceasta, reducând astfel complexitatea modelului
- Ex.:



  - Clasa *SocialSecurity* nu are comportament propriu netrivial şi nici asocieri cu alte clase, exceptând *Person*
  - Astfel de transformări trebuie amânate până spre finalul fazei de proiectare/începutul fazei de implementare, atunci când responsabilităţile claselor sunt clare

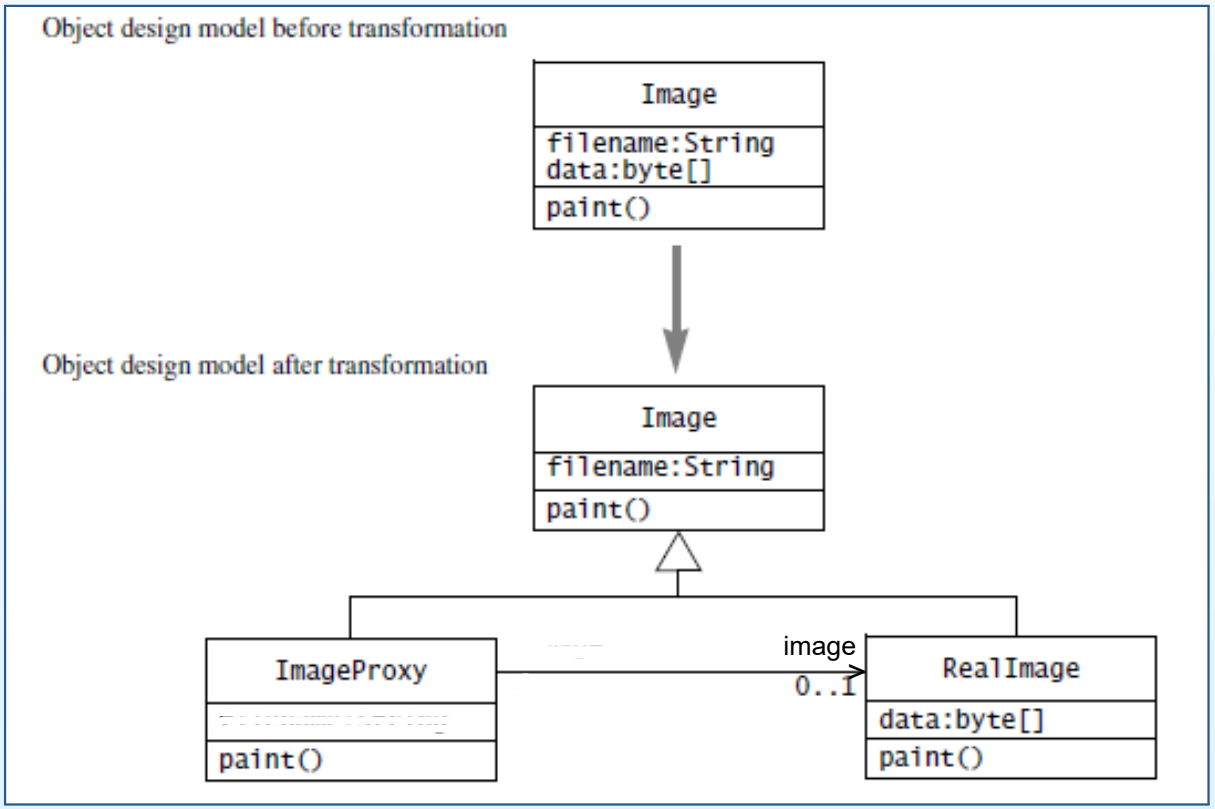
Transformarea unor clase în atribute (cont.)

- Acestei transformări îi corespunde unu caz particular al refactorizării *Inline Class* [Fowler, 2000]

- Paşi refactorizării *Inline Class*
  - Declară toate câmpurile şi metodele publice ale clasei sursă în clasa destinaţie
  - Schimbă toate referinţele spre clasa sursă către clasa destinaţie
  - Schimbă numele clasei sursă, pentru a identifica eventualele referinţe nemodificate
  - Compilează şi testează
  - Şterge clasa sursă

Gestionare operaţiilor costisitoare

- Operaţiile costisitoare, de tipul încărcării unor obiecte grafice, pot fi amânate până în momentul în care este necesară vizualizarea acestora
  - Soluţie: aplicarea şablonului *Proxy*
  - Ex.:



Gestionare operaţiilor costisitoare (cont.)

- Un obiect de tip *ImageProxy* (imagine surogat) ia locul obiectului *RealImage* (imagine reală), oferind aceeaşi interfaţă cu acesta
- Obiectul surogat răspunde la solicitări simple şi încarcă obiectul real doar în momentul în care i se apelează operaţia `paint()` (mesajul de desenare va fi apoi delegat obiectului imagine real)
- În cazul în care clienţii nu apelează `paint()`, obiectul imagine real nu va fi creat niciodata

- Rezultatele unor operaţii complexe, apelate frecvent, însă bazate pe valori care nu se schimbă sau se schimbă destul de rar, pot fi memorate la nivelul unor atribute private
  - Soluţia optimizează timpul de răspuns la apelul operaţiilor, însă consumă spaţiu de memorie suplimentar pentru stocarea unor informaţii redundante

Reprezentarea asocierilor

- UML: *asocieri* = mulţimi de legături între obiecte
- Limbaje de programare: *referinţe* / *colecţii de referinţe*
- Implementarea asocierilor în cod ţine cont de *navigabilitate*, *multiplicităţi*, *nume de roluri* şi de semantica domeniului
  - Bidirecţionalitatea introduce dependenţe mutuale între clase (se traduce prin perechi de referinţe ce trebuie sincronizate)
  - multiplicitatea *one* necesită o referinţă, cea *many* - o colecţie de referinţe
  - numele de roluri corespund numelor de câmpuri adăugate în clase

Asocieri unidirecţionale *one-to-one*

- Model

- Cod sursă după transformare

```
public class Advertiser {

    private Account account;

    public Advertiser()
    {
        account = new Account();
    }

    public Account getAccount()
    {
        return account;
    }

    // nu ofera setter
}
```

Asocieri bidirecţionale *one-to-one*

- Model

- Cod sursă după transformare

```
public class Advertiser {

    /* The account field is initialized
       in the constructor
       and never modified */
    private Account account;

    public Advertiser()
    {
        account = new Account(this);
    }

    public Account getAccount()
    {
        return account;
    }
}
```

```
public class Account {

    /* The owner field is initialized
       in the constructor
       and never modified. */
    private Advertiser owner;

    public Account(Advertiser owner)
    {
       this.owner = owner;
    }

    public Advertiser getOwner()
    {
        return owner;
    }
}
```

Asocieri bidirecţionale *one-to-many*

- Model

- Cod sursă după transformare

```
public class Advertiser {

    private Set<Account> accounts;

    public Advertiser()
    {
        accounts = new HashSet();
    }

    public void addAccount(Account account)
    {
        if(!accounts.contains(account))
        {
            accounts.add(account);
            account.internalSetOwner(this);
        }
    }
```

```
public class Account {

   private Advertiser owner;

   public void setOwner(Advertiser owner)
   {
       Advertiser oldOwner = this.owner;
       Advertiser newOwner = owner;
       if(oldOwner != null)
           oldOwner.internalRemoveAccount(this);
       if(newOwner != null)
           newOwner.internalAddAccount(this);
       this.owner = newOwner;
   }
```

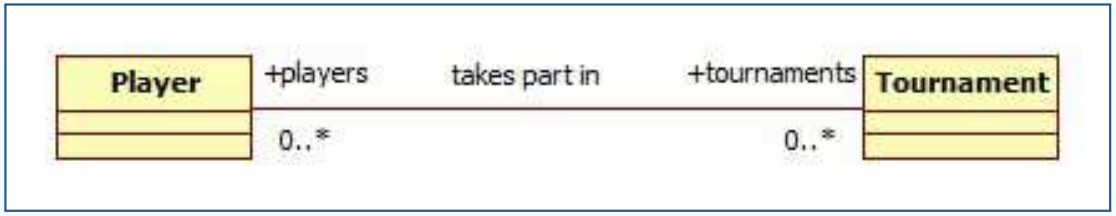
Asocieri bidirecţionale *one-to-many* (cont.)

```
    public void removeAccount(Account account)
    {
        if(accounts.contains(account))
        {
            accounts.remove(account);
            account.internalSetOwner(null);
        }
    }

   void internalAddAccount(Account account)
   {
        if(!accounts.contains(account))
            accounts.add(account);
   }

   void internalRemoveAccount(Account account)
   {
        if(accounts.contains(account))
            accounts.remove(account);
   }

   public Set<Account> getAccounts()
   {
       return Collections.unmodifiableSet(accounts);
   }

}
```

```
   void internalSetOwner(Advertiser owner)
   {
        this.owner = owner;
   }

   public Advertiser getOwner()
   {
        return owner;
   }
}
```

Asocieri bidirecţionale *many-to-many*

- Model



- Cod sursă după transformare

```
public class Player {

    private Set<Tournament> tournaments;

    public Player()
    {
        tournaments = new HashSet();
    }

    public void addTournament(Tournament tournament)
    {
        //pre: tournament != null
        if(!tournaments.contains(tournament))
        {
          tournaments.add(tournament);
          tournament.internalAddPlayer(this);
        }
    }
```

```
public class Tournament {

    private Set<Player> players;

    public Tournament()
    {
        players = new HashSet();
    }

    public void addPlayer(Player player)
    {
        //pre: player != null
        if(!players.contains(player))
        {
          players.add(player);
          player.internalAddTournament(this);
        }
    }
```

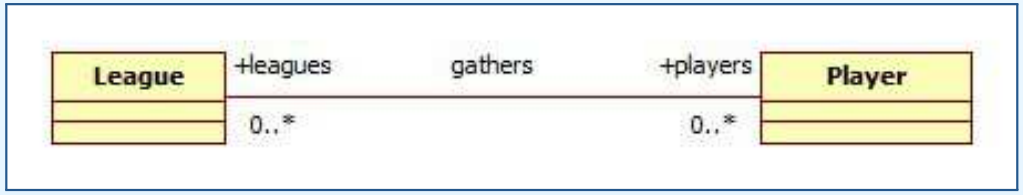
Asocieri bidirecţionale *many-to-many* (cont.)

```
    public void removeTournament(Tournament tournament)
    {
        //pre: tournament != null
        if(tournaments.contains(tournament))
        {
          tournaments.remove(tournament);
          tournament.internalRemovePlayer(this);
        }
    }

    void internalAddTournament(Tournament tournament)
    {
        //pre: tournament != null
        if(!tournaments.contains(tournament))
          tournaments.add(tournament);
    }

    void internalRemoveTournament(Tournament tournament)
    {
        //pre: tournament != null
        if(tournaments.contains(tournament))
          tournaments.remove(tournament);
    }

    public Set<Tournament> getTournaments()
    {
        return Collections.unmodifiableSet(tournaments);
    }
}
```

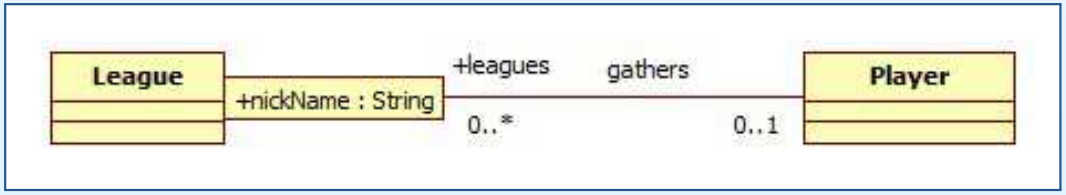
```
    public void removePlayer(Player player)
    {
        //pre: tournament != null
        if(players.contains(player))
        {
          players.remove(player);
          player.internalRemoveTournament(this);
        }
    }

    void internalAddPlayer(Player player)
    {
        //pre: tournament != null
        if(!players.contains(player))
          players.add(player);
    }

    void internalRemovePlayer(Player player)
    {
        //pre: tournament != null
        if(players.contains(player))
          players.remove(player);
    }

    public Set<Player> getPlayers()
    {
        return Collections.unmodifiableSet(playe
    }

}
```

Asocieri calificate

- Asocierile calificate sunt utilizate pentru a "reduce" multiplicitatea unui capăt *many* din cadrul unei asocieri *one-to-many* sau *many-to-many*
  - Calificatorul asocierii este un atribut al clasei din capătul *many* care se doreşte a fi redus, atribut care are valori unice în contextul asocierii, însă nu neapărat unice la nivel global
  - Ex.: Pentru a putea fi uşor identificaţi în cadrul unei ligi, jucătorii îşi pot alege un *nickName* care trebuie să fie unic în cadrul ligii (jucătorii pot avea *nickName*-uri diferite în ligi diferite, iar fiecare astfel de *nickName* nu trebuie să fie unic la nivel global)
  - Modelul înainte de transformare



  - Modelul după transformare



Asocieri calificate (cont.)

```
public class League {

    private Map<String, Player> players;

    public League()
    {
        players = new HashMap();
    }

    public void addPlayer(String nickName, Player player)
    {
        if(!players.containsKey(nickName))
        {
            players.put(nickName, player);
            player.internalAddLeague(this);
        }
    }

    public Player getPlayer(String nickName)
    {
        return players.get(nickName);
    }

    void internalAddPlayer(String nickName, Player player)
    {
        if(!players.containsKey(nickName))
            players.put(nickName, player);
    }
}
```

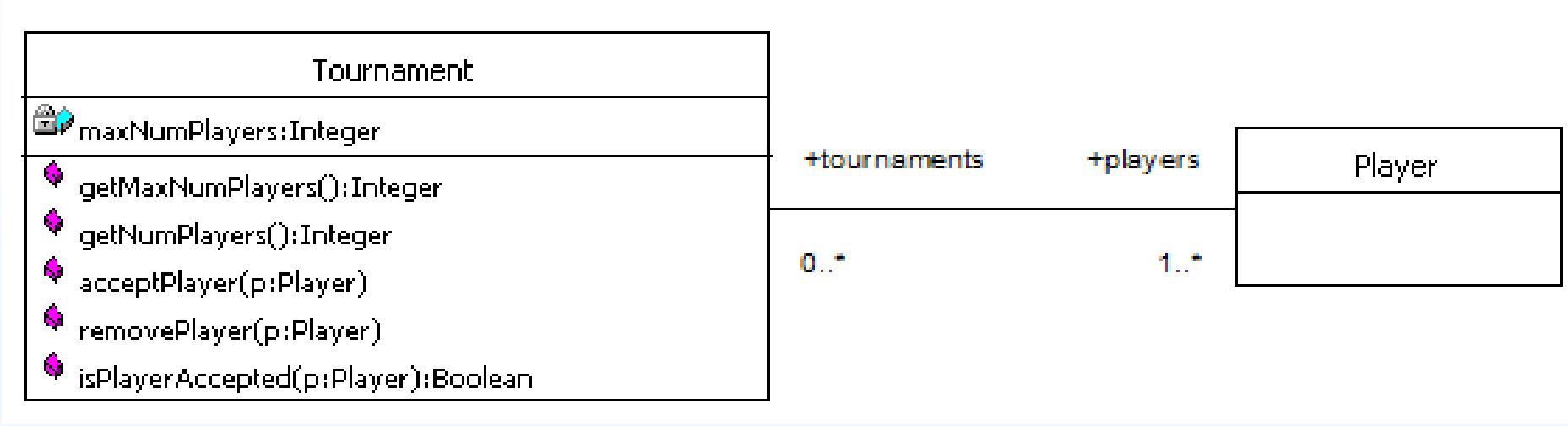
Asocieri calificate (cont.)

```
public class Player {

    private Set<League> leagues;

    public Player()
    {
        leagues = new HashSet();
    }

    public void addLeague(League league, String nickName)
    {
        if(league.getPlayer(nickName) == null)
        {
            leagues.add(league);
            league.internalAddPlayer(nickName, this);
        }
    }

    void internalAddLeague(League league)
    {
        leagues.add(league);
    }


}
```

Reprezentarea contractelor

- *Verificarea precondiţiilor*
  - Precondiţiile trebuie verificate la începutul fiecărei metode, înaintea efectuării procesărilor caracteristice. În cazul în care precondiţia nu este adevărată, se va arunca o excepţie. Se recomandă ca fiecare precondiţie să corespundă unui tip particular de excepţie.
- *Verificarea postcondiţiilor*
  - Postcondiţiile trebuie verificate la sfârşitul fiecărei metode, după terminarea tuturor procesărilor caracteristice şi finalizarea schimbărilor de stare. În cazul în care contractul este violat, se va arunca o excepţie specifică.
- *Verificarea invarianţilor*
  - Invarianţii se vor verifica odată cu postcondiţiile (la finalizarea fiecărei metode publice a clasei)
- *Moştenirea contractelor*
  - Codul de verificare al aserţiunilor trebuie încapsulat la nivelul unor metode specifice, pentru a permite apelarea acestora din subclase

Ex.: contract OCL




```
context Tournament
    inv maxNumPlayersPositive:
        self.getMaxNumPlayers() > 0

context Tournament::acceptPlayer(p:Player)
    pre:  self.getNumPlayers() < self.getMaxNumPlayers() and
          not self.isPlayerAccepted(p)
    post: self.isPlayerAccepted(p) and
          self.getNumPlayers() = self@pre.getNumPlayers() + 1
```

Ex.: implementarea contractului

```
public class Tournament {
//...
    private List players;

    public void acceptPlayer(Player p)
        throws KnownPlayer, TooManyPlayers, UnknownPlayer,
            IllegalNumPlayers, IllegalMaxNumPlayers
    {
        // check precondition!isPlayerAccepted(p)
        if (isPlayerAccepted(p)) {
            throw new KnownPlayer(p);
        }
        // check precondition getNumPlayers() < maxNumPlayers
        if (getNumPlayers() == getMaxNumPlayers()) {
            throw new TooManyPlayers(getNumPlayers());
        }
        // save values for postconditions
        int pre_getNumPlayers = getNumPlayers();
```

Ex.: implementarea contractului (cont.)

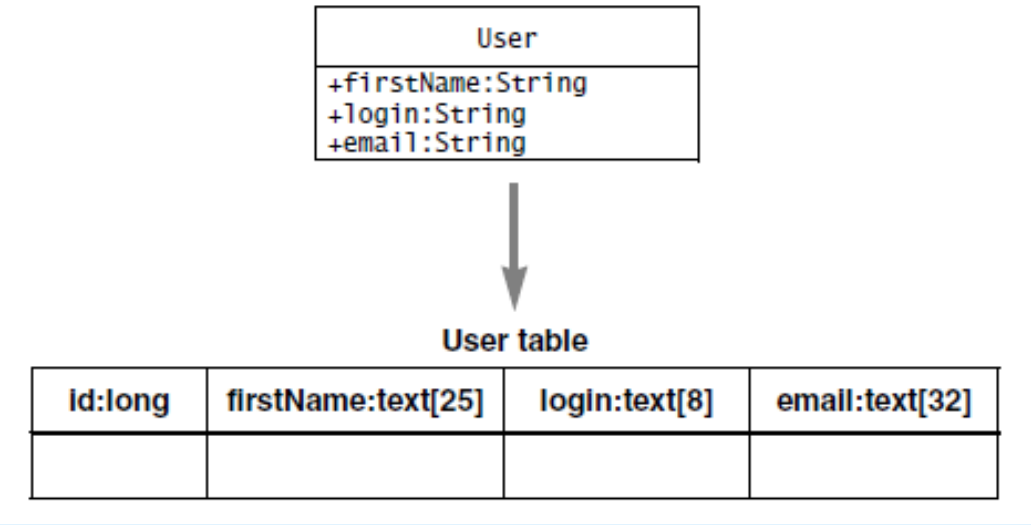
```
        // accomplish the real work
        players.add(p);
        p.addTournament(this);

        // check post condition isPlayerAccepted(p)
        if (!isPlayerAccepted(p)) {
            throw new UnknownPlayer(p);
        }
        // check post condition getNumPlayers() = @pre.getNumPlayers() + 1
        if (getNumPlayers() != pre_getNumPlayers + 1) {
            throw new IllegalNumPlayers(getNumPlayers());
        }
        // check invariant maxNumPlayers > 0
        if (getMaxNumPlayers() <= 0) {
            throw new IllegalMaxNumPlayers(getMaxNumPlayers());
        }
    }
    //...
}
```

Reprezentarea contractelor (cont.)

- Dezavantaje ale unei implementări/monitorizări manuale exhaustive a indeplinirii contractelor
  - *Efortul de codificare* - cod de verificare uneori mai complex decât logica operaţiei în sine
  - *Şanse mari de introducere a unor erori*
  - *Posibilitatea de mascare a unor defecte în codul aferent funcţionalităţii* - în cazul în care cele două sunt scrise de către acelaşi programator
  - *Dificultatea modificării codului în cazul modificării constrângerii*
  - *Probleme de performanţă* la monitorizarea exhaustivă
- *Soluţii*
  - Generarea automată a codului de verificare aferent contractelor folosind instrumente CASE dedicate (ex. OCLE)
  - Monitorizarea selectivă
    - la testare - toate aserţiunile
    - la exploatare - selectiv, funcţie de performanţele dorite, gradul de încredere în calitatea codului şi natura critică a aplicaţiei

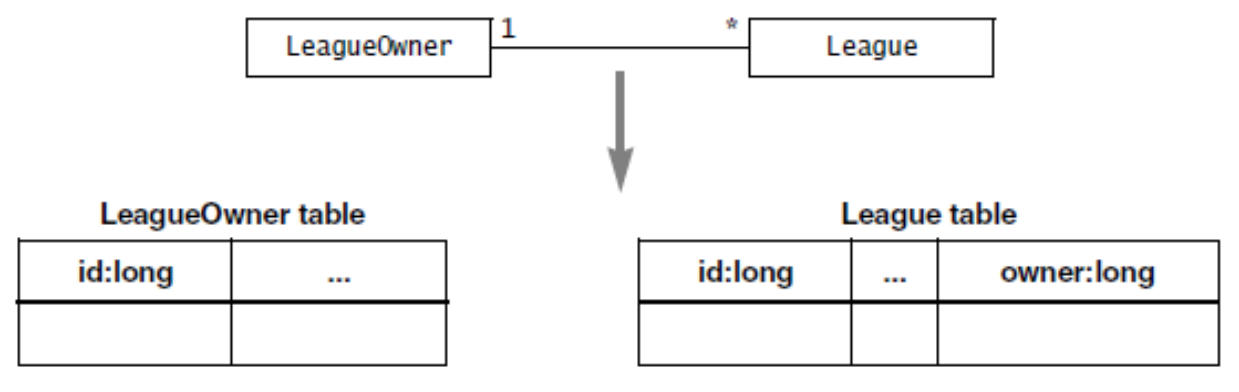
Reprezentarea entităţilor persistente

- *Reprezentarea claselor şi atributelor*
  - Fiecare clasă se reprezintă folosind un tabel cu acelaşi nume
  - Pentru fiecare atribut al clasei se adaugă în tabel o coloană cu acelaşi nume
  - Fiecare linie a unui tabel va corespunde unei instanţe a clasei
  - În mod ideal, cheia primară ar trebui sa fie un identificator unic (eventual autoincrement), diferit de atributele proprii ale clasei în cauză. Alegerea ca şi cheie a unui atribut (grup) caracteristice tipului de entitate este problematică în momentul în care apar modificări la nivelul domeniului aplicaţiei

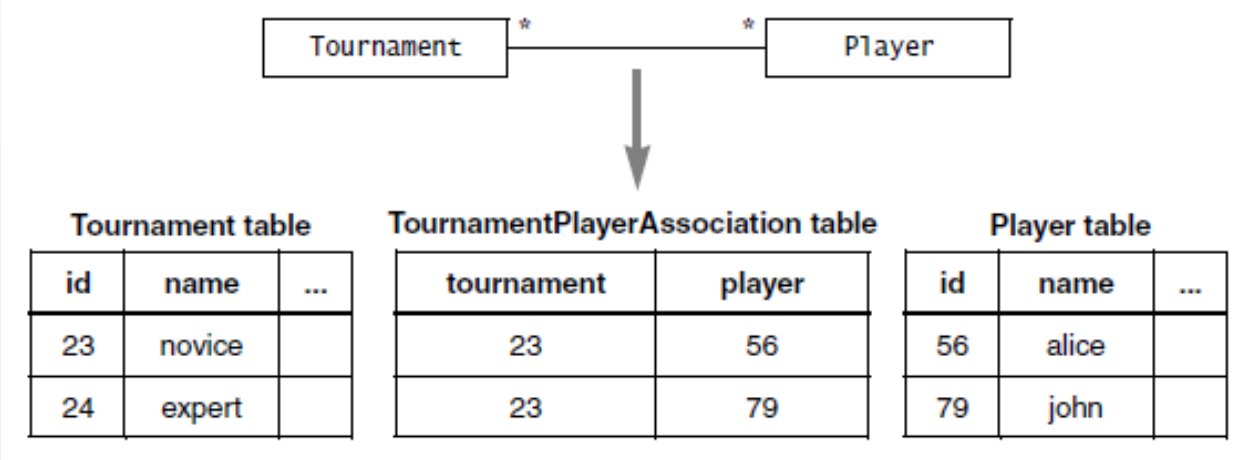


Reprezentarea entităţilor persistente (cont.)

- *Reprezentarea asocierilor*
  - Asocierile *one-to-one* şi *one-to-many* se reprezintă folosind chei străine

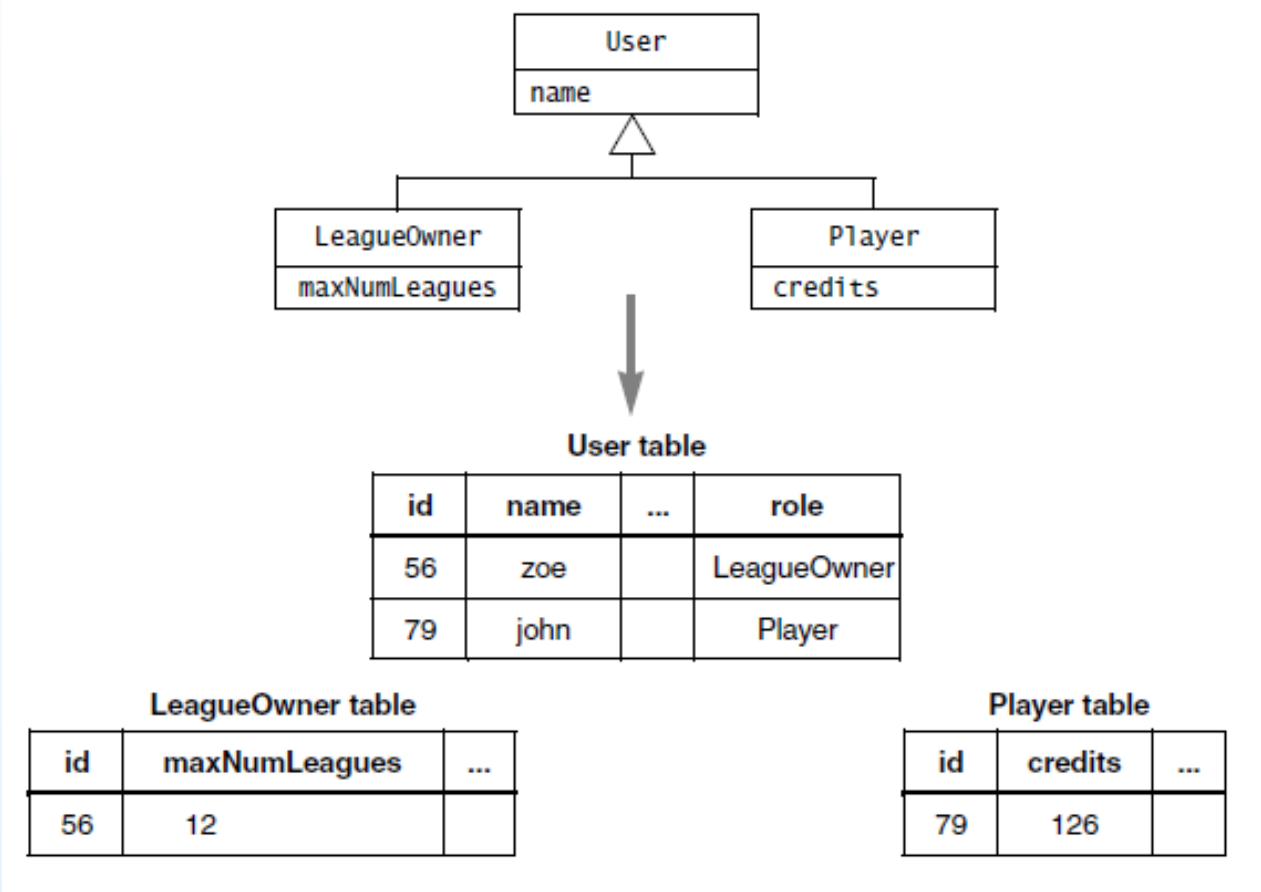


  - Asocierile *many-to-many* se reprezintă folosind tabele de legătură



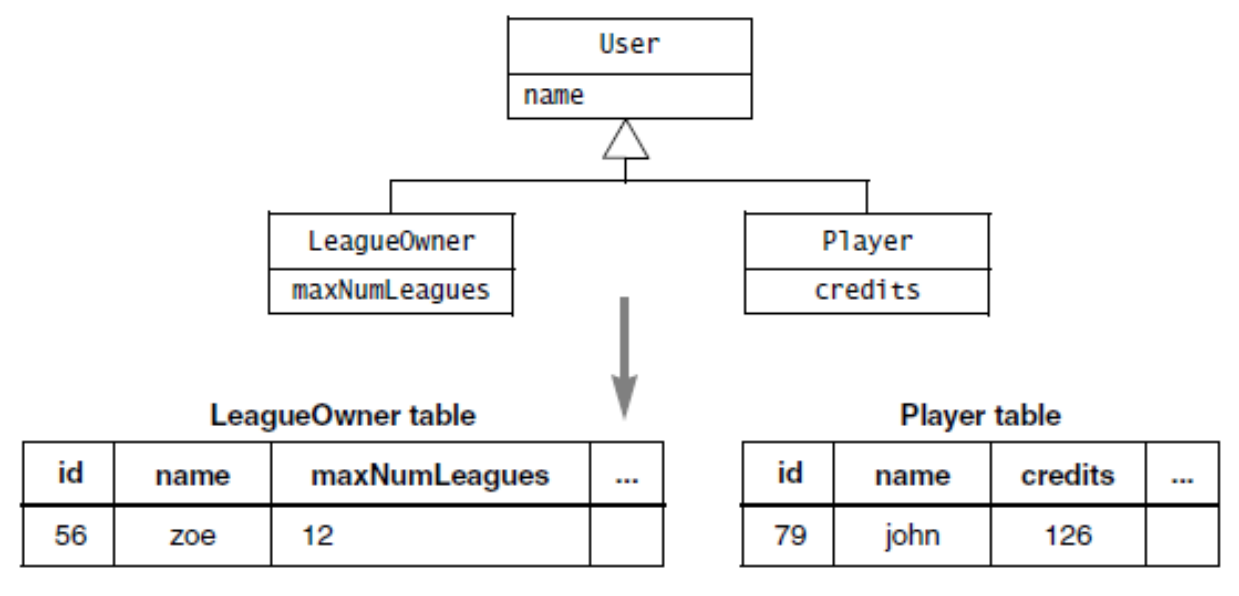
Reprezentarea entităţilor persistente (cont.)

- *Reprezentarea moştenirii*
  - Mapare verticală



Reprezentarea entităţilor persistente (cont.)

- Mapare orizontală



- Mapare verticală vs. mapare orizontală = modificabilitate vs. performanţă
  - Maparea verticală: adăugarea unui atribut în clasa de bază => adăugarea unei coloane în tabelul aferent; adăugarea unei noi clase derivate => definirea unui tabel cu atributele proprii ale acesteia; fragmentarea obiectelor individuale => interogări mai lente

Reprezentarea entităţilor persistente (cont.)

- Maparea orizontală: adăugarea unui atribut în clasa de bază => adăugarea unei coloane în fiecare dintre tabelele aferente claselor derivate; adăugarea unei noi clase derivate => definirea unui tabel cu atributele proprii + cele moştenite; nefragmentarea obiectelor individuale => interogări rapide

Referinţe

- [Fowler, 2000] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Weslwy Reading, MA, 2000.

Curs 11

Testarea sistemelor soft

Suport de curs bazat pe ***B. Bruegge and A.H. Dutoit***

"Object-Oriented Software Engineering using UML, Patterns, and Java"

Testarea sistemelor soft

- *Testarea* = procesul de identificare a diferenţelor dintre comportamentul dorit/aşteptat al sistemului (specificat cu ajutorul modelelor) şi comportamentul observat al acestuia
  - *Testarea unitară* - identifică diferenţe dintre specificarea unui obiect şi implementarea acestuia ca şi componentă
  - *Testarea structurală* - identifică diferenţe dintre modelul aferent proiectării de sistem şi comportamentul unei grup de subsisteme integrate
  - *Testarea funcţională* - identifică diferenţe dintre modelul cazurilor de utilizare şi sistem
  - *Testarea performanţei* - identifică diferenţe dintre cerinţele nefuncţionale şi performanţele sistemului
- Din perspectiva modelării, testarea reprezintă o încercare de a demonstra că implementarea sistemului este inconsistentă cu modelele acestuia
  - Scopul este proiectarea unor teste care să pună în evidenţă defectele sistemului

Fiabilitatea sistemelor soft

- *Corectitudinea* unui sistem în raport cu specificaţia - vizează concordanţa dintre comportamentul observat al sistemului şi specificarea acestuia.
  - *Fiabilitatea softului* = probabilitatea ca acel soft să nu cauzeze eşecul sistemului, pentru o anumită perioadă de timp şi în condiţii specificate [IEEE Std. 982.2-1988]
- *Eşec* (eng. *failure*) = orice deviere a comportamentului observat de la cel specificat/aşteptat
- *Eroare/stare de eroare* (eng. *erroneous state*) = orice stare a sistemului în care procesările ulterioare ar conduce la eşec
- *Defect* (eng. *fault*/*defect*/*bug*) = cauza mecanică sau algoritmică a unei stări de eroare
- *Testare* = încercarea sistematică şi planificată de a găsi defecte în softul implementat
  - "*Testing can only show the presence of bugs, not their absence.*" (E. Dijkstra)

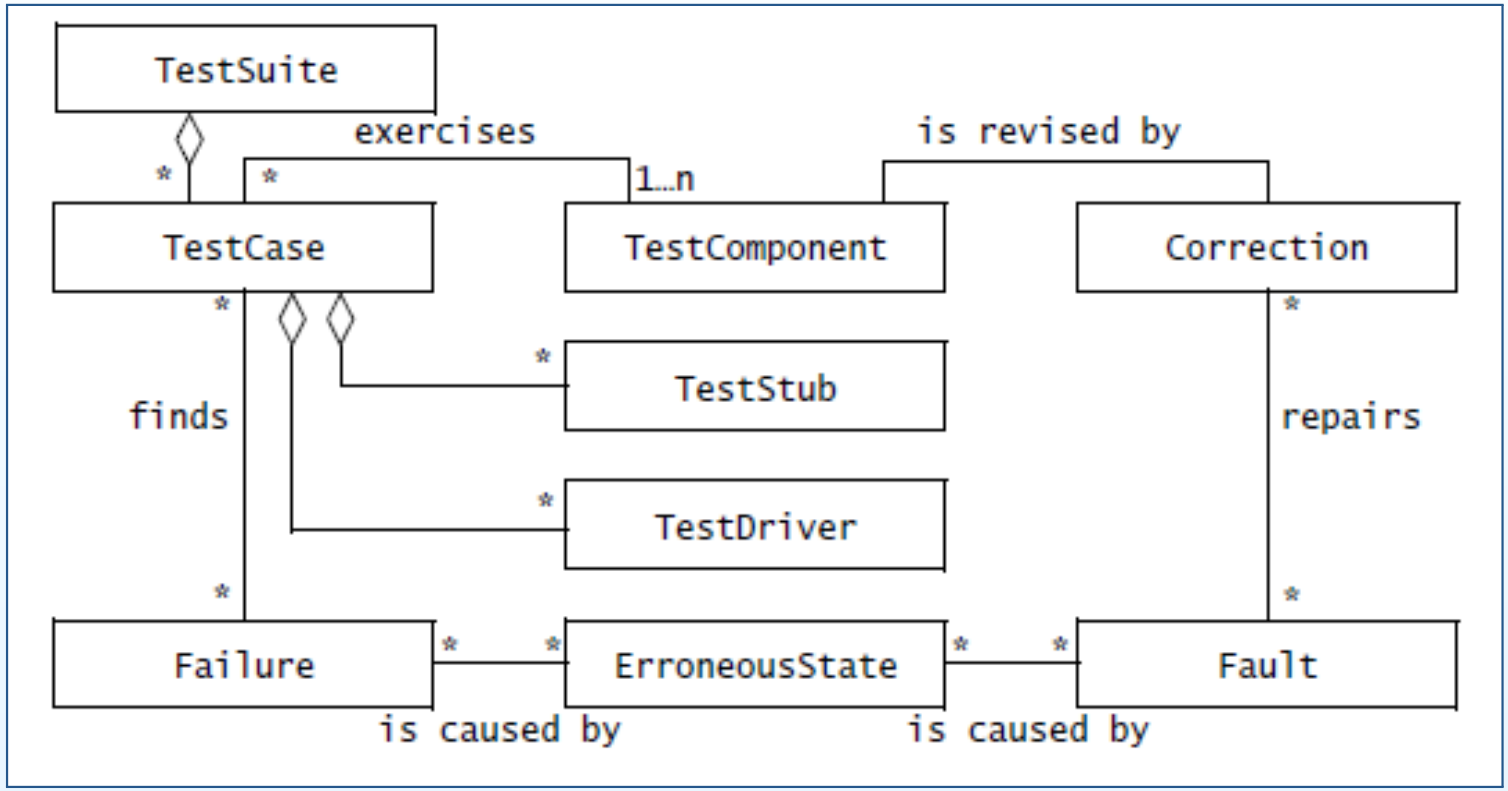
Fiabilitatea sistemelor soft (cont.)

- Tehnici de creştere a fiabilităţii sistemelor soft
  - *Tehnici privind evitarea defectelor* (eng. *fault avoidance techniques*)
    - identifică posibilele defecte la nivel static (fără o execuţie a modelelor/codului) şi încearcă să prevină introducerea acestora în sistem
    - ex.: metodologii formale - Cleanroom, Correctness by Construction
  - *Tehnici privind detectarea defectelor* (eng. *fault detection techniques*)
    - sunt utilizate în timpul procesului de dezvoltare, pentru a identifica stările de eroare şi defectele care le-au provocat, anterior livrării sistemului (ex. review, testare, depanare)
    - nu îşi propun recuperarea sistemului din stările de eşec induse de defectele identificate
  - *Tehnici privind tolerarea defectelor* (eng. *fault tolerance techniques*)
    - pornesc de la premisa că sistemul poate fi livrat cu defecte şi că eventualele eşecuri pot fi gestionate prin recuperare în urma lor la execuţie
    - ex.: sistemele redundante utilizează mai multe calculatoare şi softuri diferite pentru realizarea aceloraşi sarcini

Testarea sistemelor soft - concepte

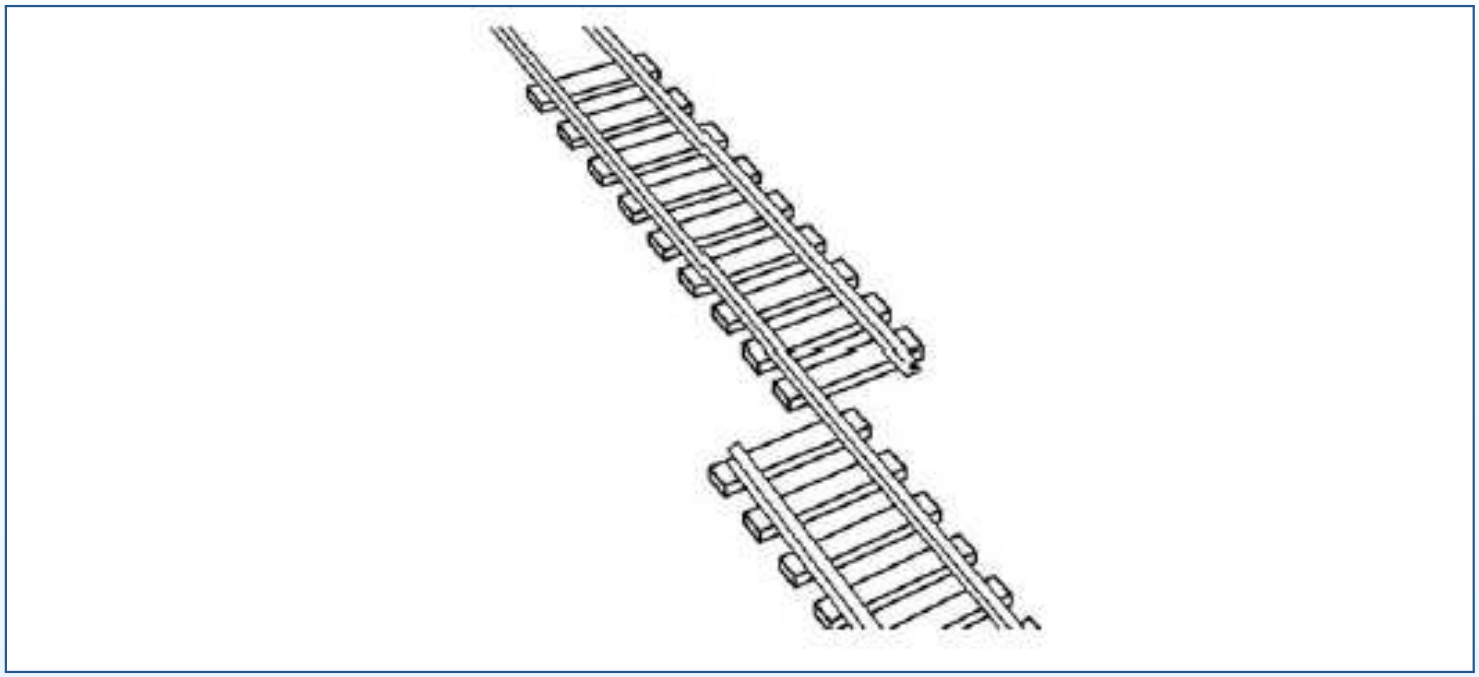
- *Componentă de test* (eng. *test component*) = parte a unui sistem care poate fi izolată pentru testare (obiect/subsistem, grup de obiecte/subsisteme)
- *Defect* (eng. *fault, bug, defect*) = greşeală de proiectare sau codificare ce poate determina un comportament anormal al unei componente
- *Stare de eroare* (eng. *error state*) = manifestare a unui defect în timpul execuţiei sistemului
- *Eşec* (eng. *failure*) = deviere a comportamentului observat al componentei de la cel specificat
- *Caz de test* (eng. *test case*) = o mulţime de date de intrare şi rezultate aşteptate, proiectate cu intenţia de a provoca eşecul sistemului şi a descoperi defecte la nivelul componentelor
- eng. *Test stub* = implementare parţială a unei componente, de care depinde componenta de test
- eng. *Test driver* = implementare parţială a unei componente care depinde de componenta de test (împreună cu stub-urile, permit izolarea unei componente pentru testare)
- *Corectură* (eng. *Correction*) = modificare a unei componente, în scopul de a remedia un defect (poate introduce noi defecte)

Testarea sistemelor soft - concepte (cont.)



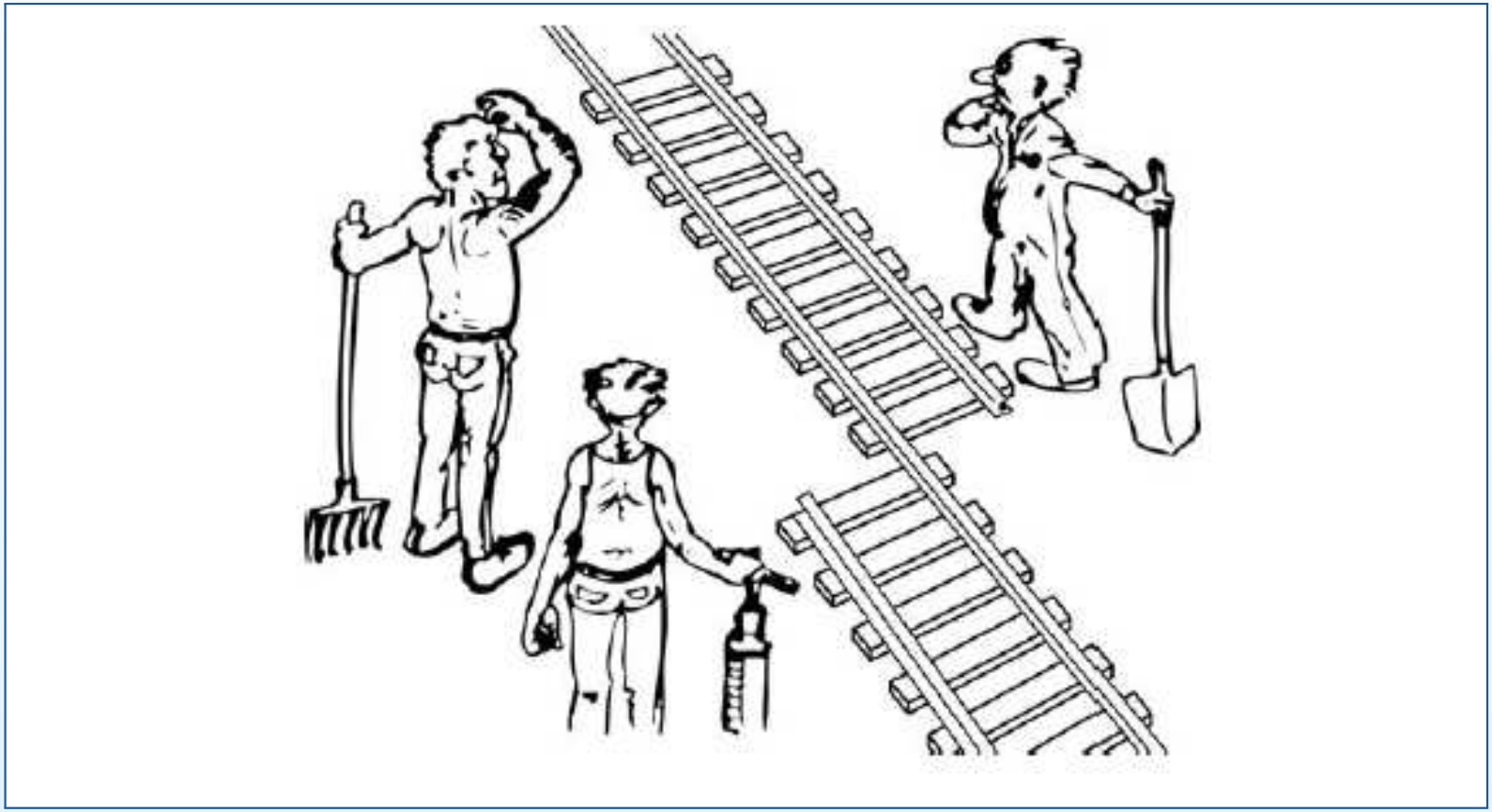
Testarea sistemelor soft - concepte (cont.)

- Eşec, eroare sau defect?



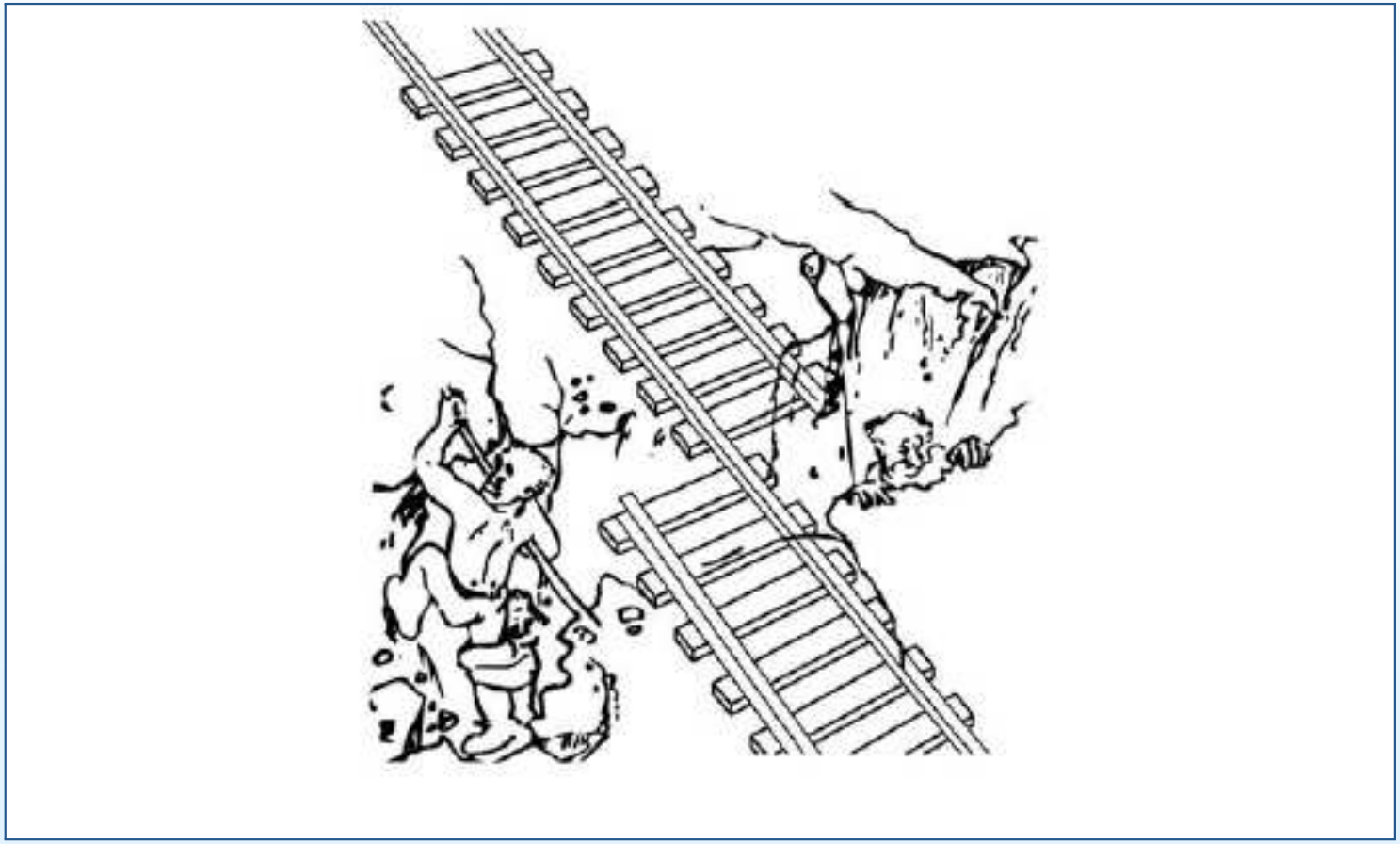
Testarea sistemelor soft - concepte (cont.)

- Defect algoritmic
  - omiterea iniţializării unei variabile
  - setarea unei variabile folosite drept index in afara valorilor permise



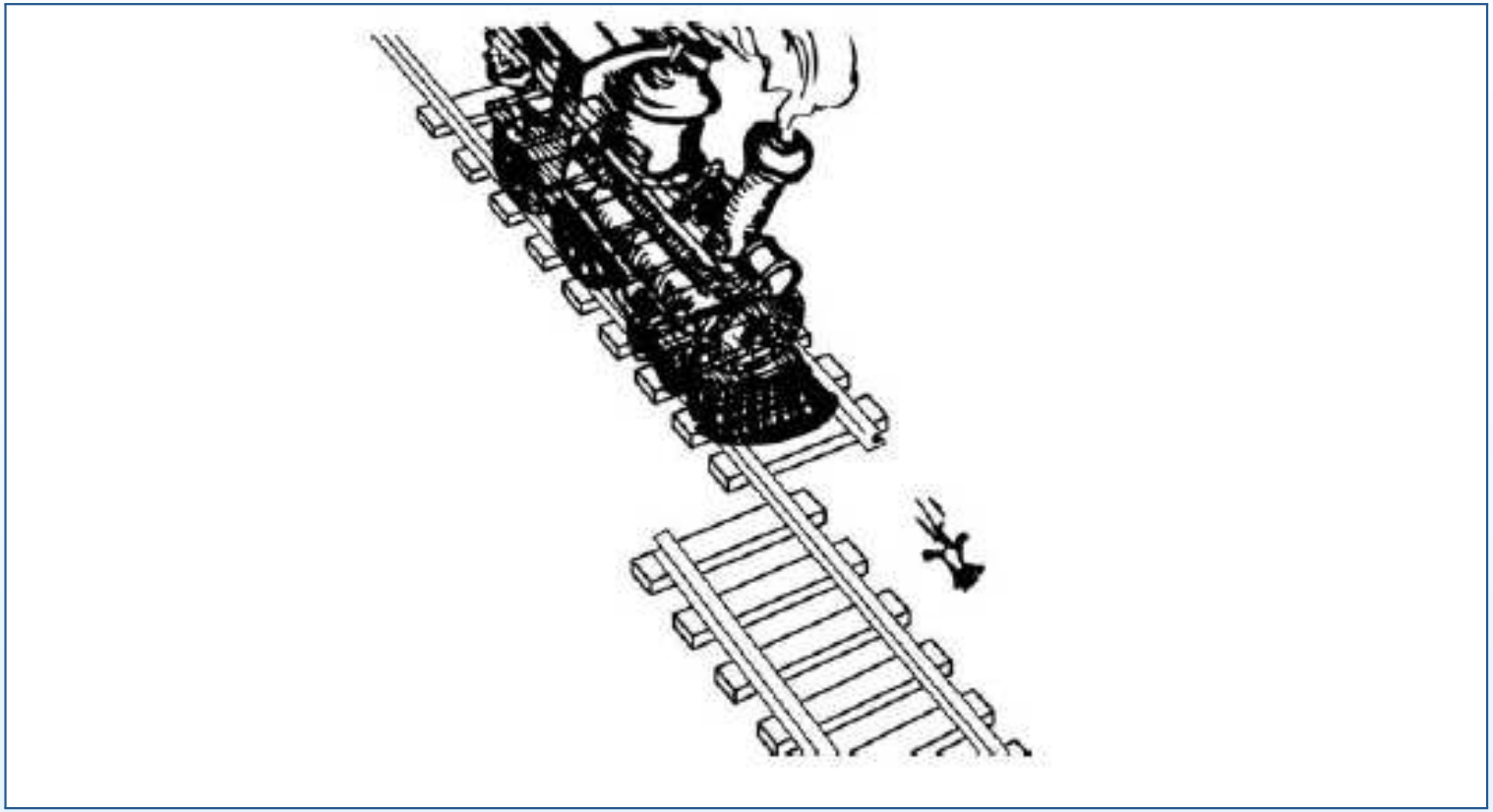
Testarea sistemelor soft - concepte (cont.)

- Defect mecanic
  - defect la nivelul maşinii virtuale
  - pană de curent



Testarea sistemelor soft - concepte (cont.)

- Eroare



Testarea sistemelor soft - activităţi

- *Planificarea testării* (eng. *test planning*)
  - alocă resurse şi programează activităţile de testare
  - această activitate ar trebui să aibă loc devreme în procesul de dezvoltare, astfel încât testării să i se aloce suficient timp şi resursă umană calificată (ex. cazurile de test ar trebui proiectate imediat ce modelele aferente devin stabile)
- *Inspectarea componentelor* (eng. *component inspection*)
  - Identifică defecte la nivelul componentelor prin inspectarea manuală a codului sursă al acestora
- *Testarea utilizabilităţii* (eng. *usability testing*)
  - încearcă să detecteze defecte în proiectarea interfeţei utilizator a sistemului
  - uneori, sistemele eşuează din cauză că utilizatorii sunt induşi în eroare de interfaţa grafică şi introduc neintenţionat date eronate
- *Testarea unitară* (eng. *unit testing*)
  - încearcă să detecteze defecte la nivelul obiectelor sau subsistemelor individuale, raportându-se la specificarea acestora (modelul aferent proiectării obiectuale)

Testarea sistemelor soft - activităţi (cont.)

- *Testarea de integrare* (eng. *integration testing*)
  - încearcă să identifice defecte prin integrarea diferitor componente, raportându-se la modelul aferent proiectării de sistem
  - *testarea structurală* (eng. *structural testing*) = testare de integrare implicând toate componentele sistemului
- *Testarea de sistem* (eng. *system testing*)
  - testează sistemul integrat în ansamblu
  - *testarea funcţională* (eng. *functional testing*) - realizată de către dezvoltatori, testează sistemul în raport cu modelul său funcţional
  - *testarea performanţei* (eng. *performance testing*) - realizată de către dezvoltatori, testează sistemul în raport cu specificarea cerinţelor nefuncţionale şi cu obiectivele adiţionale de proiectare
  - *testarea de acceptare* şi *testarea de instalare* (eng. *acceptance testing*, *installation testing*) - realizate de către clienţi în mediul de dezvoltare, respectiv de exploatare a sistemului

Inspectarea componentelor

- Inspectarea identifică defectele unei componente prin analiza codului acesteia, în cadrul unei reuniuni formale
  - Inspecţiile pot avea loc anterior sau ulterior testării unitare
- *Metoda inspectării a lui Fagan* [Fagan, 1976] - primul proces structurat de inspectare
  - Inspecţia este condusă de către o echipă de dezvoltatori, incluzând autorul componentei, un moderator şi unul sau mai mulţi recenzori
  - Paşi
    - *Overview* - autorul componentei prezintă, pe scurt, scopul acesteia şi obiectivele inspecţiei
    - *Pregătire* - recenzorii se familiarizează cu codul componentei (fără a se focusa pe identificarea defectelor)
    - *Şedinţa de inspectare* - unul dintre cei prezenţi parafrazează codul sursă al componentei şi membrii echipei semnalează probleme cu privire la acesta
    - *Revizuire* - autorul revizuieşte codul componentei, conform observaţiilor primite
    - *Urmări* - moderatorul verifică varianta revizuită şi poate stabili necesitatea de reinspectare

Inspectarea componentelor (cont.)

- Etape critice: pregătirea şi şedinţa de inspectare
- În afara identificării defectelor, recenzorii pot semnala abateri de la standardele de codificare sau ineficienţe
- Eficienţa unei inspecţii depinde de pregătirea recenzorilor

- *Active Design Review* [Parnas and Weiss, 1985] - proces de inspectare îmbunătăţit
  - Elimină şedinţa de inspectare, recenzorii identifică defecte în faza de pregătire
  - La finalul etapei de pregătire, fiecare recenzor completează un chestionar care atestă gradul de înţelegere a componenetei
  - Autorul colectează feedback asupra componentei în urma unor întâlniri individuale cu fiecare recenzor
- Ambele metode de inspectare s-au dovedit a fi mai eficiente în descoperirea defectelor decât testarea
  - În proiectele critice se recurge atât la inspecţii, cât şi la testare, întrucât au tendinţa de a identifica tipuri diferite de erori

Testarea utilizabilităţii

- Identifică diferenţele dintre sistem şi aşteptările utilizatorilor cu privire la comportamentul acestuia (spre deosebire de celelalte tipuri de testare, nu compară sistemul cu o specificaţie)
- Tehnica de realizare a testelor privind utilizabilitatea
  - Dezvoltatorii formulează un set de obiective descriind informaţia pe care se aşteaptă să o obţină în urma testelor (ex.: evaluarea layout-ului interfeţei grafice, evaluarea impactului pe care timpul de răspuns îl are asupra eficienţei utilizatorilor, evaluarea măsurii în care documentaţia online răspunde nevoilor utilizatorilor)
  - Obectivele anterioare sunt evaluate într-o serie de experimente în care reprezentaţi ai utilizatorilor sunt antrenaţi să execute anumite sarcini
  - Dezvoltatorii observă participanţii şi colectează date privind performanţele acestora (timp de îndeplinire a unei sarcini, rata erorilor) şi preferinţele lor
- Tipuri de teste de utilizabilitate
  - Teste bazate pe scenarii
  - Teste bazate pe prototipuri (verticale sau orizontale)
  - Teste pe baza sistemului real

Testarea utilizabilităţii (cont.)

- Elemente fundamentale ale testelor de utilizabilitate
  - obiectivele de test
  - reprezentanţi ai utilizatorilor
  - mediul de lucru, real sau simulat
  - interogare extensivă a utilizatorilor de către responsabilul cu testele de utilizabilitate
  - colectare şi analiză a rezultatelor cantitative şi calitative
  - recomandări cu privire la modul de îmbunătăţire a asistemului
- Obiective de test uzuale
  - compararea a două stiluri de interacţiune utilizator
  - identificarea celor mai bune/rele funcţionalităţi într-un scenariu/prototip
  - identificarea funcţionalităţilor utilie pentru începători/experţi
  - identificarea situaţiilor care necesită help online, etc.

Testarea unitară

- Se focusează pe componentele elementare ale sistemului soft - obiecte şi subsisteme
  - Candidaţi pentru testarea unitară: toate clasele modelului obiectual şi toate subsistemele identificate în proiectarea de sistem
- Avantaje
  - Reducerea complexităţii activităţilor de testare, prin focusarea pe componente cu granularitate mică
  - Uşurinţa identificării şi corectării defectelor, ca urmare a numărului mic de componente implicate într-un test
  - Posibilitatea introducerii paralelismului în activitatea de testare (componentele pot fi testate independent şi simultan)
- Tehnici de testare unitară
  - *Testarea bazată pe echivalenţe* (eng. *equivalence testing*)
  - *Testarea frontierelor* (eng. *boundary testing*)
  - *Testarea căilor de execuţie* (eng. *path testing*)
  - *Testarea bazată pe stări* (eng. *state-based testing*)
  - *Testarea polimorfismului* (eng. *polymorphism testing*)

Testarea bazată pe echivalenţe

- Tehnică de testare blackbox care minimizează numărul de cazuri de test, prin partiţionarea intrărilor posibile în clase de echivalenţă şi selectarea unui caz de test pentru fiecare astfel de clasă
  - Se presupune că sistemul se comportă în mod similar pentru toţi membrii unei clase de echivalenţă => testarea comportamentului aferent unei clase de echivalenţă se poate realiza prin testarea unui singur membru al clasei
- Paşi
  - I. Identificarea claselor de echivalenţă
  - II. Selectarea intrărilor pentru test
- Criterii utilizate în stabilirea claselor de echivalenţă
  - *Acoperire*: fiecare intrare posibilă trebuie să aparţină uneia dintre clasele de echivalenţă
  - *Caracter disjunct*: o aceeaşi intrare nu poate aparţine mai multor clase de echivalenţă
  - *Reprezentare*: Dacă, prin utilizarea ca şi intrare a unui anumit membru al unei clase de echivalenţă, execuţia conduce la o stare de eroare, atunci aceeaşi stare va putea fi detectată utilizând ca şi intrare orice alt membru al clasei

Testarea bazată pe echivalenţe (cont.)

- Ex.: testarea unei metode care returnează numărul de zile dintr-o lună, date fiind luna şi anul (întregi)

```
class MyGregorianCalendar {
...
    public static int getNumDaysInMonth(int month, int year) {…}
...
}
```

- I.1 Identificarea claselor de echivalenţă pentru lună
  - clasa lunilor cu 31 de zile (1,3,5,7,8,10,12)
  - clasa lunilor cu 30 de zile (4,6,9,11)
  - clasa lunilor cu 28/29 de zile (2)
- I.2 Identificarea claselor de echivalenţă pentru an
  - clasa anilor bisecţi
  - clasa anilor non-bisecţi
- Valori invalide
  - pentru lună: < 1, > 12
  - pentru an: < 0

Testarea bazată pe echivalenţe (cont.)

- II. Selectarea reprezentanţilor pentru test
  - II.1 Pentru lună: 2 (February), 6 (June), 7 (July)
  - II.2 Pentru an: 1904, 1901
  - => 6 clase de echivalenţă prin combinaţie

Equivalence class	Value for month input	Value for year input
Months with 31 days, non–leap years	7 (July)	1901
Months with 31 days, leap years	7 (July)	1904
Months with 30 days, non–leap years	6 (June)	1901
Month with 30 days, leap year	6 (June)	1904
Month with 28 or 29 days, non–leap year	2 (February)	1901
Month with 28 or 29 days, leap year	2 (February)	1904

Testarea frontierelor

- Caz particular al metodei de testare bazată pe echivalenţe, focusată pe explorarea cazurilor limită
  - În loc să se aleagă un reprezentant arbitrar al clasei de echivalenţă pentru testare, metoda cere alegerea unui element aflat "la limită" (caz particular)
  - Presupunerea care stă la baza acestui tip de testare este aceea că dezvoltatorii omit adesea cazurile speciale ("frontierele" claselor de echivalenţă) (ex.: 0, stringuri vide, anul 2000, etc.)
- Ex.: pentru exemplul considerat anterior
  - Luna 2 (February) şi anii 1900 si 2000 (un an multiplu de 100 nu este bisect decât daca este şi multiplu de 400)
  - Lunile 0 şi 13, aflate la limita clasei de echivalenţă conţinând lunile invalide

Equivalence class	Value for month input	Value for year input
Leap years divisible by 400	2 (February)	2000
Non–leap years divisible by 100	2 (February)	1900
Nonpositive invalid months	0	1291
Positive invalid months	13	1315

Testarea căilor de execuţie

- Metodă de tip whitebox, care identifică defecte în implementarea unei componente, prin testarea tuturor căilor de execuţie din cod
  - Presupunerea din spatele acestei tehnici este aceea că, prin execuţia, cel puţin o dată, a fiecărei căi (eng. *path*) din cod, majoritatea defectelor vor genera eşecuri
  - Punctul de start în aplicarea acestei metode îl constituie construirea unei reprezentări de tip schemă logică (eng. *flow graph*) asociate codului testat
    - nodurile corespund instrucţiunilor
    - arcele corespund fluxului de control
  - Ex.: implementare greşită a metodei *getNumDaysInMonth()*

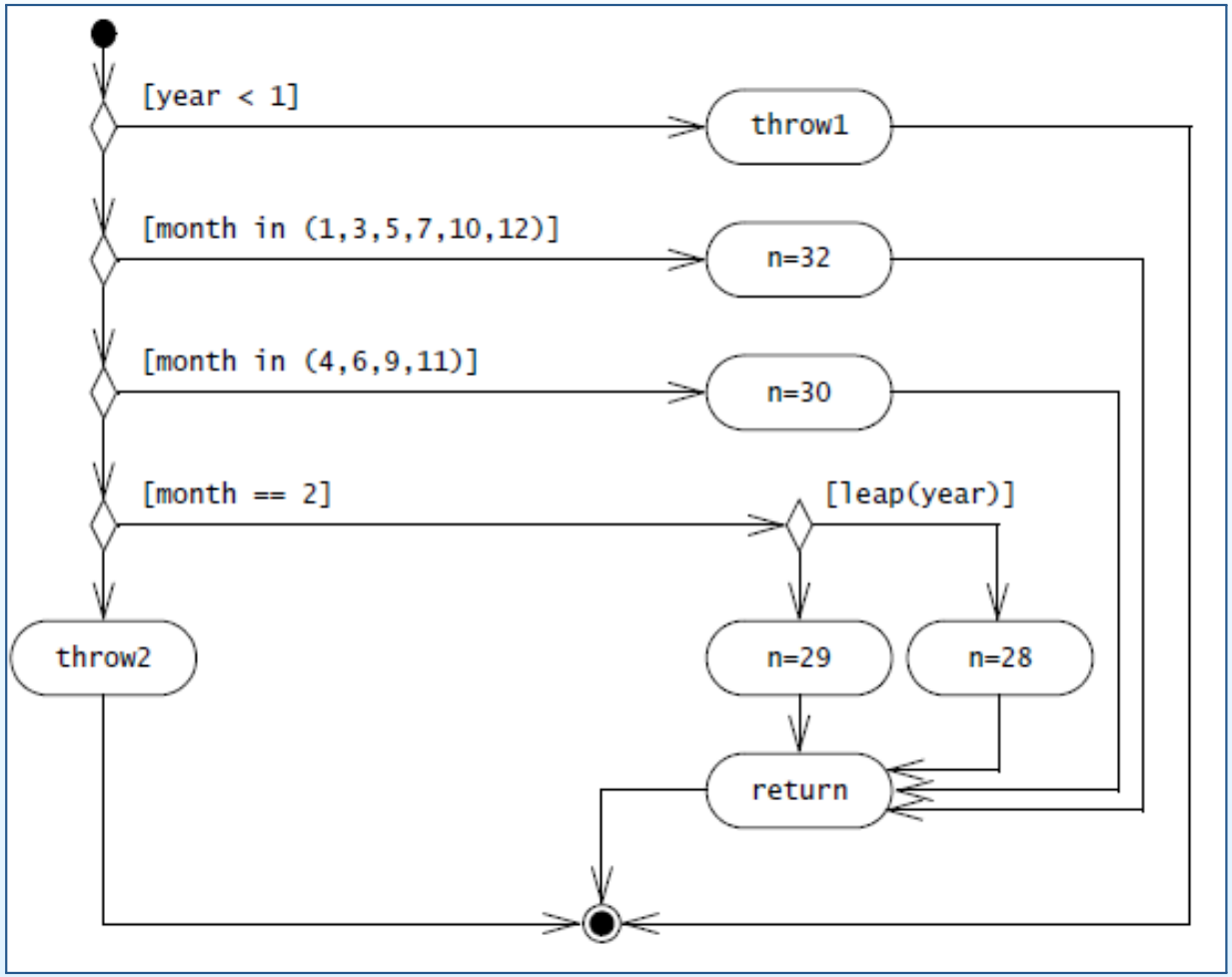
```
public class MonthOutOfBounds extends Exception {…};
public class YearOutOfBounds extends Exception {…};

class MyGregorianCalendar {
    public static boolean isLeapYear(int year) {
        boolean leap;
        if ((year%4) == 0){
            leap = true;
        } else {
            leap = false;
        }
        return leap;
    }
```

Testarea căilor de execuţie (cont.)

```
    public static int getNumDaysInMonth(int month, int year)
            throws MonthOutOfBounds, YearOutOfBounds {
        int numDays;
        if (year < 1) {
            throw new YearOutOfBounds(year);
        }
        if (month == 1 || month == 3 || month == 5 || month == 7 ||
                month == 10 || month == 12) {
            numDays = 32;
        } else if (month == 4 || month == 6 || month == 9 || month == 11) {
            numDays = 30;
        } else if (month == 2) {
            if (isLeapYear(year)) {
                numDays = 29;
            } else {
                numDays = 28;
            }
        } else {
            throw new MonthOutOfBounds(month);
        }
        return numDays;
    }
}
```

Testarea căilor de execuţie (cont.)

- Schema logică aferentă implementării (greşite a) metodei *getNumDaysInMonth()* (diagramă UML de activităţi)



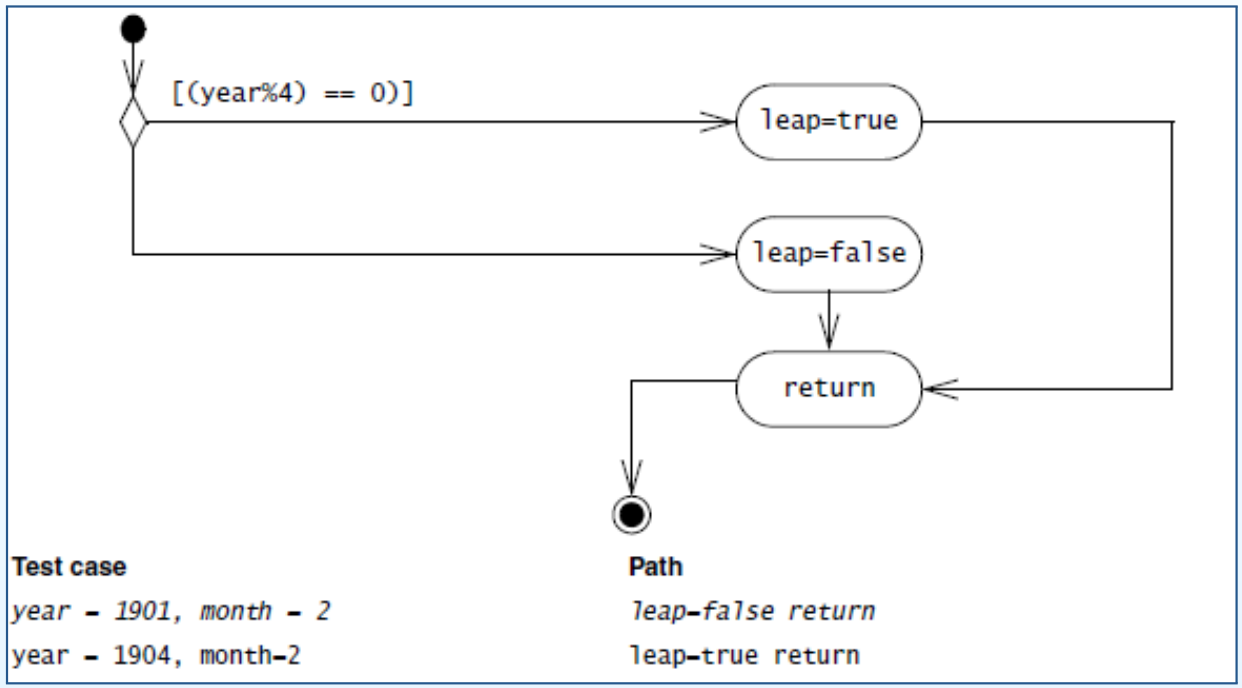
Testarea căilor de execuţie (cont.)

- Testarea completă a căilor de execuţie presupune proiectarea cazurilor de test astfel încât fiecare arc al diagramei de activităţi să fie traversat cel puţin o dată
  - Aceasta presupune analiza fiecărui nod decizional şi selectarea câte unei intrări pentru fiecare dintre ramurile *true* şi *false*
    - combinaţia (1,1901) identifică defectul *n=32*
  - Ex.: cazurile de test generate pentru metoda *getNumDaysInMonth()*

Test case	Path
(year = 0, month = 1)	{throw1}
(year = 1901, month = 1)	{n=32 return}
(year = 1901, month = 2)	{n=28 return}
(year = 1904, month = 2)	{n=29 return}
(year = 1901, month = 4)	{n=30 return}
(year = 1901, month = 0)	{throw2}

Testarea căilor de execuţie (cont.)

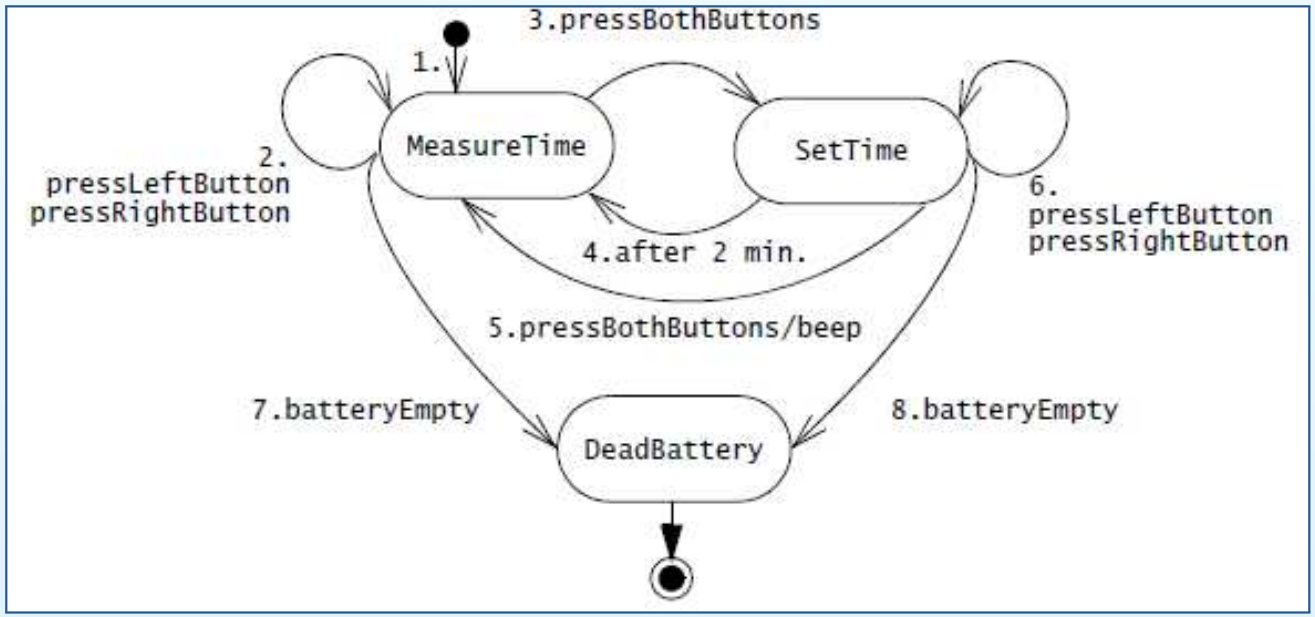
- Ex.: diagrama de activităţi şi cazurile de test generate pentru metoda *isLeapYear()*



- Testarea căilor de execuţie vs. testarea bazată pe echivalenţe / a frontierelor
  - ambele detectează defectul *n=32*
  - prima e posibil să nu genereze caz de test aferent anilor multiplu de 100
  - e posibil ca nici unul dintre cazurile de test generate de cele două metode să nu identifice eroarea asociată lunii 8

Testarea bazată pe stări

- Tehnică de testare a sistemelor orientate obiect, care generează cazuri de test pentru o clasă pe baza diagramei UML de tranziţie a stărilor asociată respectivei clase
  - Pentru fiecare stare, se stabileşte un set reprezentativ de stimuli aferenţi tranziţiilor posibile din acea stare (similar testării bazate pe echivalenţe)
  - După aplicarea fiecărui stimul, se compară starea curentă a componentei cu cea indicată de diagramă, indicându-se eşec în caz de neconcordanţă
- Ex.: diagramă de tranziţie a stărilor aferentă clasei *2Bwatch*



Testarea bazată pe stări (cont.)

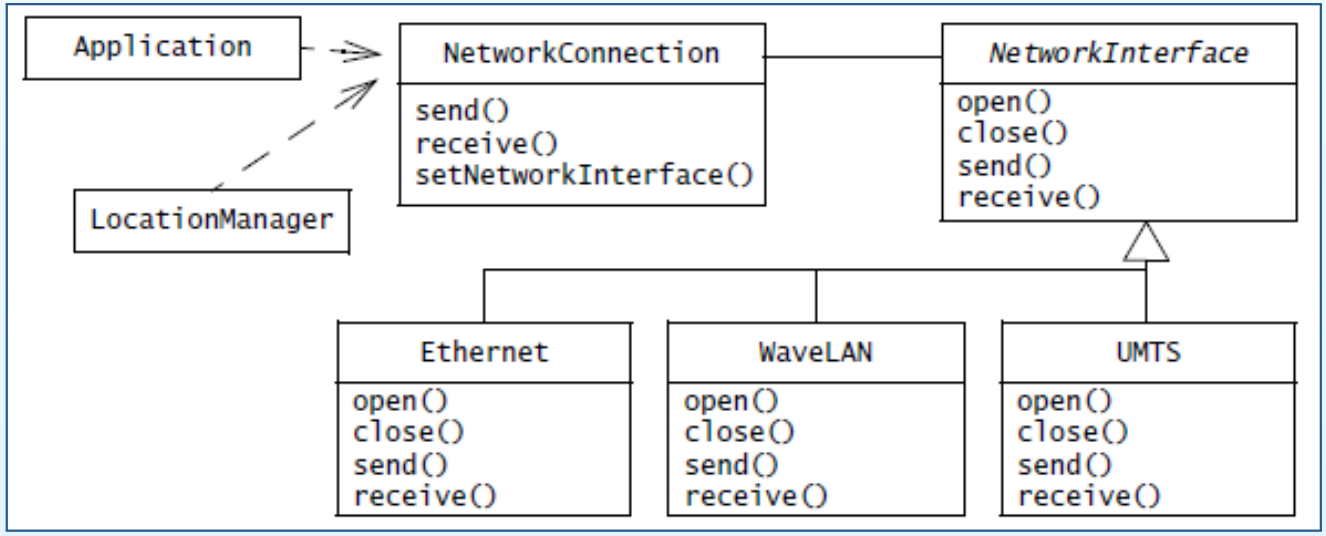
- Ex.: cazuri de test generate pentru sistemul *2Bwatch* (a.î. fiecare tranziţie, exceptând 7 şi 8, să fie traversată cel puţin o dată)

Stimuli	Transition tested	Predicted resulting state
Empty set	1. *Initial transition*	`MeasureTime`
Press left button	2.	`MeasureTime`
Press both buttons simultaneously	3.	`SetTime`
Wait 2 minutes	4. *Timeout*	`MeasureTime`
Press both buttons simultaneously	3. *Put the system into the SetTime state to test the next transition.*	`SetTime`
Press both buttons simultaneously	5.	`SetTime->MeasureTime`
Press both buttons simultaneously	3. *Put the system into the SetTime state to test the next transition.*	`SetTime`
Press left button	6. ~~Loop back onto `MeasureTime`~~	~~`MeasureTime`~~ SetTime

- Avantaje/dezavantaje ale testării bazate pe stări
  - - Starea fiind încapsulată, cazurile de test trebuie să includă aplicarea unor secvenţe de stimuli care aduc componenta în starea dorită, înainte de a putea testa o anumită tranziţie
  - + Potenţial de automatizare

Testarea polimorfismului

- Polimorfismul introduce o nouă provocare în procesul de testare, prin faptul că permite ca un acelaşi mesaj să se concretizeze în apeluri de metode diferite, funcţie de tipul actual al apelatului
  - Atunci când se realizează testarea căilor de execuţie pentru o metodă ce utilizează polimorfism, este necesar să se ia în calcul toate legăturile posibile => necesitatea de a expanda metoda pentru a aplica algoritmul clasic de test
  - Ex.: aplicare a şablonului *Strategy* pentru a încapsula diferite implementări *NetworkInterface*



Testarea polimorfismului (cont.)

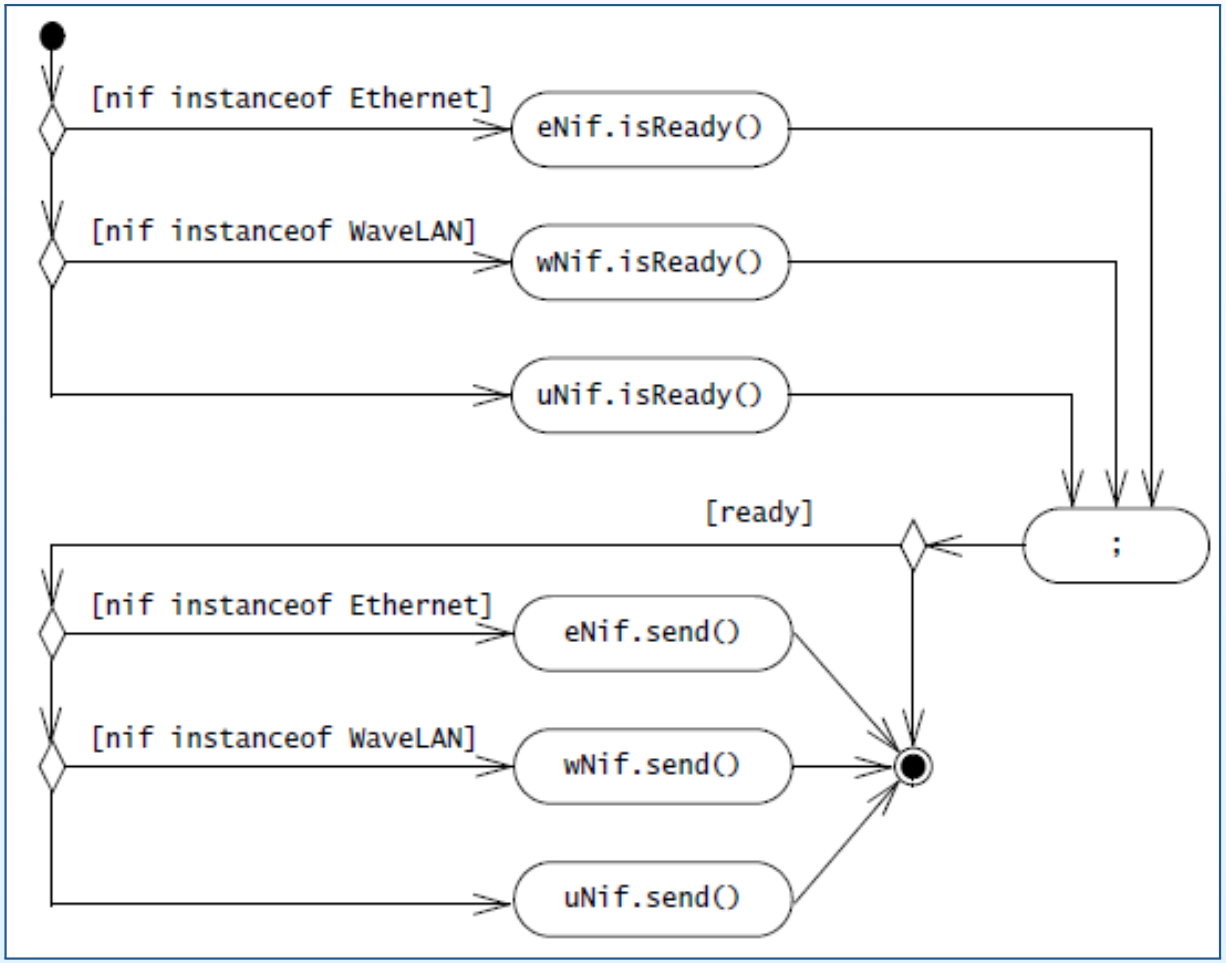
- Ex.: Codul sursă al metodei *NetworkConnection.send()*, cu şi fără polimorfism (ultima variantă este cea folosită pentru generarea cazurilor de test)

```
public class NetworkConnection {
//...
private NetworkInterface nif;
void send(byte msg[]) {
    queue.concat(msg);
    if (nif.isReady()) {
        nif.send(queue);
        queue.setLength(0);
    }
}
}
```

```
public class NetworkConnection {
//...
private NetworkInterface nif;
void send(byte msg[]) {
    queue.concat(msg);
    boolean ready = false;
    if (nif instanceof Ethernet) {
        Ethernet eNif = (Ethernet)nif;
        ready = eNif.isReady();
    } else if (nif instanceof WaveLAN) {
        WaveLAN wNif = (WaveLAN)nif;
        ready = wNif.isReady();
    } else if (nif instanceof UMTS) {
        UMTS uNif = (UMTS)nif;
        ready = uNif.isReady();
    }
    if (ready) {
        if (nif instanceof Ethernet) {
            Ethernet eNif = (Ethernet)nif;
            eNif.send(queue);
        } else if (nif instanceof WaveLAN){
            WaveLAN wNif = (WaveLAN)nif;
            wNif.send(queue);
        } else if (nif instanceof UMTS){
            UMTS uNif = (UMTS)nif;
            uNif.send(queue);
        }
        queue.setLength(0);
    }
}
}
```

Testarea polimorfismului (cont.)

- Ex.: Diagrama de activităţi aferentă variantei expandate a codului sursă al metodei *NetworkConnection.send()* (generarea cazurilor de test se face după metoda prezentată la "Testarea căilor de execuţie")



Referinţe

- [Fagan, 1976] M. E. Fagan, *Design and code inspections to reduce errors in program development*, IBM Systems Journal, Vol. 15, No. 3, 1976.
- [Parnas and Weiss, 1985] D. L. Parnas and D. M. Weiss, *Active design reviews: principles and practice*, Proceedings of the Eighth International Conference on Software Engineering, London, U.K., pp 132-136, August 1985.