Proiectare Software

Adapted after :

Timothy Lethbridge and Robert Laganiere,
Object-Oriented Software Engineering –
Practical Software Development using UML and Java, 2005
(chapter 9)

9.1 Procesul de proiectare

Definitie:

- In contextul ingineriei software, *proiectarea* este un proces de rezolvare probleme al carui obiectiv consta in descoperirea si descrierea unui mod de:
  - —implementare a *cerintelor functionale*
  - —cu respectarea constrangerilor impuse de *cerintele nonfunctionale*
  - —si a principiilor generale de *calitate*.

Optiuni de proiectare

Inginerul software trebuie sa rezolve *probleme de proiectare*.

- Pentru fiecare problema de proiectare exista mai multe solutii alternative, sau *optiuni de proiectare*.
- Rezolvarea fiecarei probleme implica *decizii de proiectare*.
  - —Este un proces de alegere a celei mai bune optiuni dintre mai multe alternative.

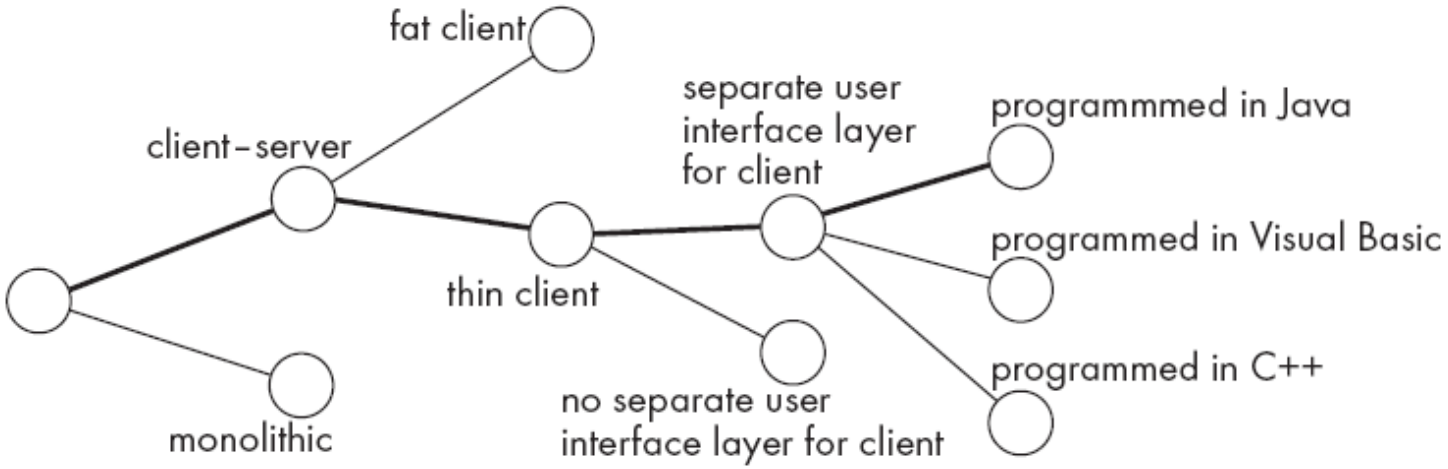
In luarea deciziilor de proiectare inginerul software ia in considerare:

- cerintele software,
- partea de proiect construita pana in acel moment,
- tehnologia disponibila,
- principii si practici de proiectare,
- experienta proiectelor anterioare.

Optiuni de proiectare

• Rareori se intampla ca doi ingineri software sa aleaga exact aceeasi solutie la o problema de proiectare.

• Figura de mai jos descrie optiunile (spatiul) de design pentru un posibil proiect software.



• Fiecare decizie tehnica ar trebui inregistrata, impreuna cu ratiunea de proiectare pe care se bazeaza.

Subsisteme, componente si module

- **_Modulul_ reprezinta ‘caramida’ de baza in modularizarea sistemelor software. In general, un modul _nu_ este un sistem independent.**
  - In cele ce urmeaza, notiunea de *modul* denota un element definit la nivelul limbajului de programare.
    - —Exemple de module:
      - metode, clase si pachete in Java
      - functii si fisiere modul in C.
- **In sens strict, o _componenta_ este o parte fizica a unui sistem, care poate fi inlocuita cu o componenta similara si care furnizeaza realizarea (si se conformeaza) unui set de interfete [BRJ99].**
- **Un _sistem_ este un ansamblu de elemente si conexiuni care alcatuiesc un tot unitar.**
- **Un _subsistem_ este o parte a unui sistem care apare de sine statator intr-un proces.**

Proiectare top-down si bottom-up

Proiectare top-down

- Se proiecteaza mai intai structura de ansmablu a sistemului.
- Se continua gradual prin decizii de proiectare tot mai detaliate.

Proiectare bottom-up

- Se incepe prin decizii de proiectare referitoare la componente reutilizabile de nivel coborat.
- Se decide apoi modul in care acestea pot fi combinate pentru construirea structurii de ansamblu a sistemului.

In practica, abordarile top-down si bottom-up sunt combinate:

- Abordarea top-down este aproape intotdeauna necesara pentru proiectarea structurii de ansamblu a sistemului
- Abordarea bottom-up permite favorizeaza reutilizarea componentelor.

9.2 Principiile activitatii de proiectare

Prin aplicarea **principiilor de proiectare** prezentate in continuare inginerul software urmareste sa obtina:

- Sporirea profitului (reducerea costurilor si cresterea veniturilor)
- Conformitatea proiectului cu cerintele
- Accelerarea procesului de dezvoltare
- Asigurarea calitatii
  - —Fiabilitate
  - —Mentenabilitate
  - —Reutilizabilitate
  - —Eficienta

Principiul 1: Divide and conquer (divide et impera)

O problema complexa poate fi mai usor abordata daca este mai intai descompusa in probleme mai simple.

- Fiecare subproblema poate fi abordata de o echipa separata.
- Fiecare inginer software se poate specializa.
- Fiecare subproblema este mai usor de inteles si abordat.

Un sistem software poate fi descompus in mai multe moduri:

- Un sistem poate fi descompus in subsisteme
- Un subsistem poate fi descompus in pachete
- Un pachet este alcatuit din clase
- O clasa este alcatuita din metode

Principiul 2: Cresterea gradului de coeziune

Un subsistem sau un modul prezinta un grad ridicat de coeziune atunci cand manifesta unitate si legatura interna puternica intre elementele componente.

- Un sistem alcatuit din module coezive este mai usor de inteles si de modificat.
- Pot fi evidentiate diverse tipuri de coeziune a modulelor software. In lista data mai jos tipurile de coeziune sunt ordonate descrescator dupa gradul de coeziune:
  - —Coeziune functionala
  - —Coeziune de nivel
  - —Coeziune de comunicare
  - —Coeziune de secventiere
  - —Coeziune procedurala
  - —Coeziune temporala
  - —Coeziune de utilitate

Coeziune functionala

Modulul efectueaza o sarcina de calcul specifica (unica) si returneaza un rezultat, fara a produce efecte laterale.

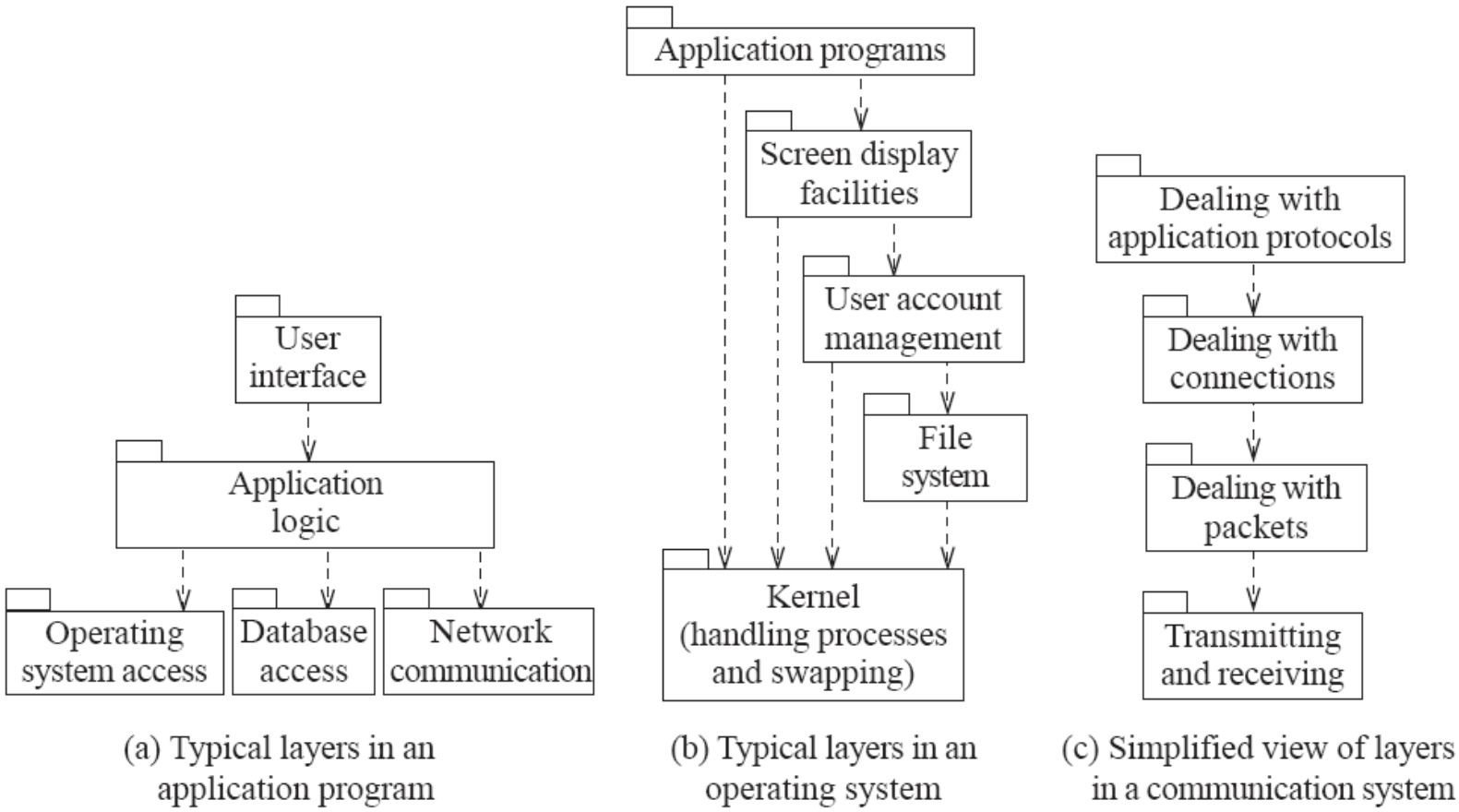
- Un modul este lipsit de efecte laterale daca nu modifica starea sistemului atunci cand efectueaza o sarcina de calcul.
  - —Un modul care actualizeaza o baza de date, interactioneaza cu utilizatorul sau creaza un fisier nou nu este functional coeziv.
- Un modul functional coeziv este:
  - —Usor de inteles
  - —Usor de reutilizat
  - —Usor de inlocuit

Coeziune de nivel (engl. *layer cohesion*)

Toate serviciile furnizate utilizatorului sau nivelelor superioare sunt grupate (functionalitatea nivelului) si orice alta functionalitate este plasata in alta parte a sistemului.

- Nivelele alcatuiesc o ierarhie.
  - —Nivelele superioare pot accesa serviciile nivelelor inferioare.
  - —Nivelele inferioare nu acceseaza nivelele superioare.
- Efectele laterale sunt permise, chiar esentiale in unele aplicatii (de exemplu intr-un nivel de acces la baza de date).
- Orice nivel poate fi inlocuit fara vreun impact asupra celorlalte nivele
  - —Este suficient sa se inlocuiasca API (engl. *Application Programming Interface*) pentru nivelul respectiv
- API = setul de proceduri prin care un nivel isi furnizeaza serviciile

Exemple



(a) Typical layers in an application program

(b) Typical layers in an operating system

(c) Simplified view of layers in a communication system

Coeziune de comunicare

Toate elementele de procesare care acceseaza sau manipuleaza anumite date sunt pastrate impreuna (de exemplu, in aceeasi clasa) si orice alta functionalitate este plasata in alta parte a sistemului.

- Unul dintre punctele forte ale paradigmei OO consta tocmai in faptul ca aceasta favorizeaza coeziunea de comunicare.
- Coeziunea de nivel nu trebuie sacrificata pentru obtinerea coeziunii de comunicare.
  - —De exemplu, chiar daca anumite obiecte (date) sunt stocate intr-o baza de date sau pe un site la distanta, o clasa ar trebui sa incarce sau sa salveze obiecte (date) doar utilizand serviciile API ale nivelelor inferioare.

Coeziune de secventiere

O serie de proceduri de calcul, care isi furnizeaza intrari si se executa in secventa, sunt pastrate impreuna si orice alta functionalitate este plasata in alta parte a sistemului.

- Coeziunea de comunicare nu ar trebui sacrificata pentru obtinerea coeziunii de secventiere.
  - —Metodele din clase diferite pot sa isi furnizeze (reciproc) intrari si pot fi apelate in secventa, dar este preferabil ca fiecare sa fie pastrata in clasa de care apartine.

Coeziune procedurala

Mai multe proceduri care se executa intr-o anumita ordine sunt pastrate impreuna, chiar daca nu isi furnizeaza intrari una alteia.

- Mai slaba decat coeziunea de secventiere

Coeziune temporala

Operatiile efectuate in aceeasi faza a executiei programului sunt pastrate impreuna si orice alta functionalitate este plasata in alta parte a sistemului.

- Exemplu: plasarea in aceeasi sectiune a codului executat in faza de initializare a sistemului
- Mai slaba decat coeziunea procedurala

Coeziune de utilitate

Utilitatile logic conectate sunt pastrate impreuna.

- Aici, o *utilitate* este o procedura sau o clasa care are aplicabilitate larga la mai multe subsisteme si este proiectata a fi reutilizabila.
- De exemplu clasa **`java.lang.Math`** are coeziune de utilitate.
  - —De remarcat faptul ca functiile din clasa **`java.lang.Math`** nu manifesta coeziune de comunicare, de secventiere, procedurala sau temporala; sunt legate intre ele doar din punct de vedere logic.

Principiul 3: Reducerea gradului de cuplare

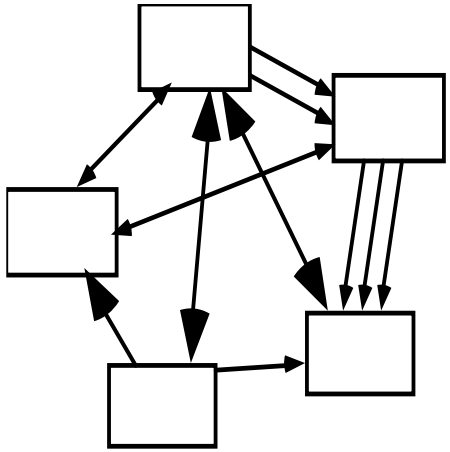
***Cuplarea* apare atunci cand exista *interdependente* intre module**

- Atunci cand exista interdependente, modificarile efectuate intr-o parte a unui sistem pot sa atraga dupa sine modificari in alta parte a sistemului.
- O retea de interdependente ingreuneaza intelegerea componentelor individuale.

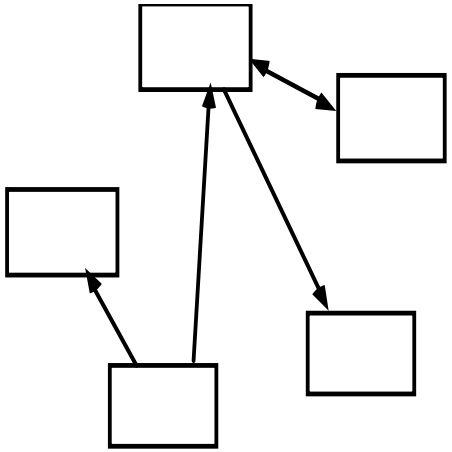
Reducerea gradului de cuplare

- Tipuri de cuplare (date in ordinea descrescatoare a gradelor de cuplare):
  - —De continut
  - —Prin date globale
  - —De control
  - —De marca
  - —Prin date
  - —Prin apel rutina
  - —Prin utilizare tip
  - —Prin import
  - —Externa

Sistem cu cuplare intensa



Sistem cu grad redus de cuplare



- Pentru reducerea gradului de cuplare trebuie sa se reduca:
  - —*numarul* conexiunilor intre module si
  - —*taria* conexiunilor (sugerata in figura prin grosimea sagetilor)

Cuplare de continut

Apare atunci cand un modul modifica *pe ascuns* date *interne* ale unui alt modul

- Cuplarea de continut ar trebui sa fie intotdeauna evitata, deoarece orice element de procesare a datelor ar trebui sa fie usor de localizat si usor de inteles.
- Pentru reducerea cuplarii de continut toate variabilele de instanta ar trebui sa fie *incapsulate*:
  - —declarate `private`
  - —accesate numai prin metode `set` si `get`
- O forma mai subtila de cuplare de continut apare atunci cand se modifica direct o variabila de instanta a unei variabile de instanta (chiar daca toate variabilele de instanta sunt private), ca in exemplul urmator:

Exemplu de cuplare de continut

```
public class Line
{
  private Point start, end;
  ...
  public Point getStart() { return start; }
  public Point getEnd()  { return end; }
}

public class Arch
{
  private Line baseline;
  ...
  void slant(int newY)
  //   Modifica coordonata y a variabilei thEnd (punct)
  //   a variabilei baseline (linie)
  {
    Point theEnd = baseline.getEnd();
    theEnd.setLocation(theEnd.getX(),newY);
  }
}
```

Cuplare prin date globale

Apare la utilizarea de variabile globale – toate modulele care utilizeaza o variabila globala devin cuplate intre ele si cu modulul care declara respectiva variabila

- In Java, variabilele `public static` servesc drept variabile globale.
- O forma mai slaba de cuplare prin date globale apare atunci cand o variabila poate fi accesata dintr-o *submultime* a claselor sistem
  - —De exemplu clasele dintr-un pachet Java
- Cuplarea prin date globale este acceptabila atunci cand:
  - —Se utilizeaza constante globale partajate de intregul sistem
  - —Se utilizeaza variabile globale deoarece este mai dificil sa se forteze ca un numar mare de rutine sa transmita anumite date sub forma de parametri.

Cuplare de control

Apare atunci cand o procedura apeleaza o alta procedura utilizand un ‘flag’ care controleaza in mod explicit logica (comportamentul) celei de a doua proceduri

—Exemplu:

```
public routineX(String command)
{
   if (command.equals("drawCircle")
   {
      drawCircle();
   }
   else
   {
      drawRectangle();
   }
}
```

• Pentru realizarea unei modificari trebuie modificata atat metoda apelanta cat si metoda apelata.
• Acolo unde este posibil, utilizarea operatiilor polimorfice reprezinta cea mai buna modalitate de reducere a cuplarii de control.

Cuplare de marca (engl. *stamp coupling*)

Apare atunci cand una dintre clasele aplicatiei este utilizata ca *tip* pentru argumentul unei metode

- Orice modificare este ingreunata de faptul ca o clasa utilizeaza o alta clasa (exista o dependenta semantica)
  - —Reutilizarea unei clase impune reutilizarea celeilalte
- Doua modalitati de reducere a cuplarii de marca:
  - —Utilizarea unei interfete ca tip al argumentului
  - —Transmiterea de date simple

Exemplu de cuplare de marca

```
public class Emailer
{
  public void sendEmail(Employee e, String text)  {...}
  ...
}
```

• Problema este ca metoda `sendMail()` *nu are nevoie* sa acceseze intreg obiectul `Employee`; este suficient sa acceseze numele si adresa de email a unui `Employee`.

Doua moduri de reducere a cuplarii de marca

Utilizarea unei interfete ca tip al argumentului metodei:

```
public interface Addressee {
  public abstract String getName();
  public abstract String getEmail();
}

public class Employee implements Addressee   {…}

public class Emailer {
  public void sendEmail(Addressee e, String text)   {...}
  ...
}
```

Transmiterea de date simple (inlocuieste cuplarea de marca cu cuplare prin date):

```
public class Emailer {
  public void sendEmail(String name, String email, String text)   {...}
  ...
}
```

Cuplare prin date

Apare atunci cand tipurile argumentelor unei metode sunt fie primitive fie clase de biblioteca (nu ale aplicatiei)

- Cu cat o metoda are mai multe argumente cu atat cuplarea este mai tare
  - —Toate metodele care o utilizeaza trebuie sa transmita toate argumentele
- Cuplarea prin date ar trebui redusa evitand transmiterea de argumente inutile catre metode.
- Evident, metodele trebuie sa aiba argumente, deci un anumit grad de cuplare de marca sau prin date exista in orice aplicatie netriviala.

Cuplare prin apel rutina

Apare atunci cand o rutina (o metoda intr-un sistem OO) apeleaza o alta rutina

- Rutinele sunt cuplate deoarece comportamentul uneia depinde de comportamentul celeilalte.
- Cuplarea prin apel de rutina este prezenta in orice sistem netrivial.
- Intr-un program in care o secventa de doua sau mai multe metode este utilizata pentru un anumit calcul in mod repetat, cuplarea prin apel de rutina poate fi redusa prin scrierea unei singure rutine care incapsuleaza respectiva secventa.

Cuplare prin utilizare tip

Apare atunci cand un modul utilizeaza un tip de date definit intr-un alt modul

- Apare oricand o clasa declara o variabila de instanta sau o variabila locala avand ca tip o alta clasa.
- In scopul reducerii acestui gen de cuplare, pentru tipul unei variabile se alege cea mai generala clasa sau interfata care contine operatiile necesare.

Cuplare prin import

Apare atunci cand un modul importa un alt modul (de exemplu un pachet in Java)

- Modulul care importa depinde semantic de modulul importat.
- Daca in modulul importat se modifica sau se adauga ceva, atunci poate aparea un conflict (de exemplu de nume) cu modulul care realizeaza operatia de import; acesta din urma trebuie la randul sau sa fie modificat pentru solutionarea conflictului.
  - —In general, cuplarea prin import nu poate fi evitata, deoarece aceasta permite utilizarea bibliotecilor sistem.
  - —Desigur, este indicat sa se evite importarea de pachete sau clase care nu sunt necesare in aplicatie.

Cuplare externa

Apare atunci cand un modul are o dependenta semantica fata de sistemul de operare, pachete software produse de o alta firma, componente hardware, etc.

- Corespunde unui nivel relativ ridicat de cuplare (mai redus decat cuplarea prin date globale, dar mai ridicat decat cuplarea de control, cf. [Pre97]).
- Este indicat sa se reduca numarul de locuri din cod in care exista asemenea dependente.
- Sablonul de design Façade poate reduce acest tip de cuplare prin simplificarea interfetei spre functiile externe.

Principiul 4: Mentinerea unui nivel cat mai inalt de abstractizare

Pentru reducerea complexitatii, este important ca in fiecare etapa de proiectare sa se mentina un nivel cat mai ridicat de abstractizare, prin ascunderea sau amanarea tratarii detaliilor.

- Mentinerea unui nivel ridicat de abstractizare permite inginerului software
  - —Sa se concentreze asupra fiecarei probleme la un anumit nivel de generalitate,
  - —Sa ignore detaliile irelevante.
- Abstractizarea este necesara deoarece creierul uman poate procesa o cantitate limitata de informatie in fiecare moment.
  - —Limbajul UML poate fi utilizat pentru modelarea sistemelor software la diferite nivele de abstractizare [BRJ99].
    - Diagramele UML pot vizualiza modele cu diferite grade de detaliere.
  - —Abstractiile procedurale sau de date sunt de obicei suportate direct la nivelul limbajului de programare.

Principiul 5: Sporirea gradului de reutilizabilitate

Diferitele aspecte ale unui sistem software ar trebui proiectate astfel incat sa poata fi utilizate din nou in diferite contexte, in cadrul sistemului curent sau in cadrul altor sisteme.

• Reutilizabilitatea poate fi incorporata la nivel de procedura, clasa, framework

—Un *framework* este un sablon arhitectural, extensibil, specific unui domeniu de aplicatie [BRJ99, JBR99].

• Strategii de baza pentru sporirea gradului de reutilizabilitate (construirea de componente reutilizabile):

—Construirea unui proiect cat mai simplu si cat mai general

—Aplicarea precedentelor trei principii de proiectare (cresterea gradului de coeziune, reducerea gradului de cuplare, sporirea nivelului de abstractizare)

Principiul 6: Reutilizare design sau cod

Reutilizarea anumitor parti de proiect sau de cod reprezinta beneficiul obtinut in urma efortului investit in construirea de componente reutilizabile.

- *Clonarea codului* (adica copierea anumitor segmente de cod dintr-o parte in alta) nu ar trebui sa fie considerata o forma de reutilizare.

Principiul 7: Proiectare pentru flexibilitate

Proiectarea pentru *flexibilitate* (sau *adaptabilitate*) este anticiparea eventualelor modificari ale proiectului si pregatirea pentru acestea.

- Moduri de realizare a unui proiect flexibil:
  - —Reducerea gradului de cuplare si cresterea gradului de coeziune a modulelor
  - —Crearea de entitati abstracte
  - —Pastrarea optiunilor de proiectare
    - Nu ar trebui limitate optiunile de proiectare pentru cei ce vor modifica ulterior sistemul
  - —Construirea de cod reutilizabil si reutilizarea codului

Principiul 8: Anticiparea uzurii

Anticiparea uzurii unui produs software se realizeaza prin masuri de planificare care tin cont de evolutia tehnologiei sau a mediului de lucru astfel incat produsul software sa continue si in viitor sa poata fi utilizat sau sa poata fi modificat cu usurinta.

- Pentru anticiparea uzurii unui produs software:
  - —Se vor evita versiunile mai vechi ale tehnologiei software / hardware
  - —Se va evita utilizarea de biblioteci specifice unor medii de lucru particulare
  - —Se vor evita serviciile nedocumentate sau putin utilizate ale bibliotecilor software
  - —Se va evita utilizarea de software sau hardware specializat furnizat de companii despre care se crede ca nu manifesta stabilitate pe termen lung
  - —Se vor utiliza limbaje si instrumente standard disponbile prin mai multi furnizori

Principiul 9: Proiectare pentru portabilitate

Proiectarea pentru portabilitate are drept obiectiv realizarea de produse software care sa poata fi utilizate pe cat mai multe platforme de calcul.

- Pentru realizarea unui produs portabil se va evita utilizarea de facilitati specifice unei platforme particulare
    - De exemplu, o biblioteca software disponibila numai sub Microsoft Windows.

Principiul 10: Proiectare pentru testabilitate

In faza de proiectare a unui produs software se pot lua masuri pentru inlesnirea activitatii de testare.

- Este avantajos ca proiectul sa fie realizat astfel incat sa faciliteze *testarea automata* a produsului software.
  - —Intreaga functionalitate ar trebui sa poata fi executata fara trecere printr-o interfata utilizator (IU) grafica.
    - In acest scop se separa IU de nucleul functional al aplicatiei
      - » O alta solutie ar fi sa se construiasca o versiune de tip linie de comanda a sistemlui.
    - Se pot apoi elabora componente de testare care apeleaza serviciile API ale nivelului functional.

Proiectare pentru testabilitate

- In fiecare clasa se pot scrie metode utilizate ca drivere de test pentru celelalte metode ale clasei.
  - In Java, se poate crea o metoda `main()` (driver de test) pentru verificarea celorlalte metode ale clasei.

—Pentru asigurarea faptului ca o subclasa se comporta in mod consistent cu superclasa sa, driverele de test din superclasa trebuie sa fie executate in fiecare subclasa.

Principiul 11: Proiectare defensiva

Inginerii software trebuie sa ia masuri de precautie (sa 'proiecteze defensiv') pentru a preintampina utilizarea necorespunzatoare a componentelor pe care le construiesc.

- Asta inseamna ca trebuie verificata validitatea intrarilor componentelor.
  - —Practic, trebuie verificate *preconditiile* fiecarei componente.
  - —In proiectarea defensiva ar trebui sa se evite verificarile repetate in mod inutil.

Proiectare prin contract

***Proiectarea prin contract* este o tehnica ce ofera un cadru pentru proiectarea defensiva sistematica si eficienta.**

- Ideea de baza
  - —Fiecare metoda are un *contract* explicit cu utilizatorii sai
- Pentru fiecare metoda, contractul consta din asertiuni care stabilesc:
  - —*preconditiile* (solicitate de metoda inaintea executiei),
  - —*postconditiile* (pe care metoda le garanteaza la sfarsitul executiei) si
  - —*invariantii* (pe care metoda se angajeaza sa nu ii modifice in timpul executiei).

9.5 Arhitectura software

Architectura software **se refera la structura de ansamblu a sistemului software si la modul in care aceasta structura furnizeaza sistemului integritate semantica [SG96].**

- Procesul de proiectare arhitecturala consta in:
  - —Descompunerea sistemului software in subsisteme
  - —Determinarea interfetelor subsistemelor
  - —Decizii privind modul de interactiune intre subsisteme
- Documentatia produsa ca rezultat al procesului de proiectare arhitecturala este adesea numita *model arhitectural*.
- Arhitectura este nucleul proiectului software; fiecare inginer software trebuie sa o inteleaga.
  - —Orice decizie nepotrivita luata in faza de proiectare a arhitecturii software poate avea implicatii negative asupra eficientei, reutilizabilitatii si mentenabilitatii sistemului.

Importanta arhitecturii software

Pentru participantii la un proiect software, dezvoltarea unui model arhitectural inlesneste:

- Intelegerea sistemului
- Organizarea procesului de dezvoltare
- Sporirea gradului de reutilizabilitate
- Evolutia sistemului

Continutul unui model arhitectural

Arhitectura unui sistem este adesea exprimata prin mai multe *vederi*

- Descompunerea logica a sistemului in subsisteme cu interfetele lor
- Dinamica interactiunii intre subsisteme si componente (la executie)
- Datele partajate intre subsisteme
- Componentele si masinile de calcul pe care componentele sunt desfasurate la executie

Observatii:

—O descriere de arhitectura arata ca o descriere completa de sistem (poate include orice tip de model) dar este mai mica: contine numai modelele si vederile sistem relevante din punct de vedere arhitectural [JBR99].

—Un model arhitectural descrie atat aspecte structurale cat si aspecte comportamentale ale sistemului.

Stabilitatea arhitecturii software

Pentru asigurarea fiabilitatii si mentenabilitatii unui sistem software, modelul arhitectural trebuie sa fie stabil.

- O arhitectura software este *stabila* daca permite adaugarea de noi servicii si facilitati la sistem cu modificari minime la nivelul arhitecturii.
- De exemplu, in UP (the Unified Process [JBR99]), prin definitie, la sfarsitul fazei de *elaborare* arhitectura software ar trebui sa fie stabila.
  - —Succesul unui proiect UP depinde de capacitatea unei mici echipe de arhitecti si dezvoltatori (de valoare profesionala ridicata) de a produce o arhitectura stabila cu resurse limitate.
    - Conform [JBR99], este posibila dezvoltarea unei arhitecturi stabile cu doar aprox. 30% din resursele alocate proiectului.

Dezvoltarea unui model arhitectural

- **Mai intai se construieste o arhitectura tentativa (inainte de considerarea detaliilor legate de cazurile de utilizare).**
  - —In acest stadiu se utilizeaza:
    - Cunostinte despre *domeniul de aplicatie*,
    - *Sabloane arhitecturale* generale (urmeaza a fi discutate).
  - —*Sugestie*: mai multe echipe diferite pot sa lucreze in mod independent la conceperea de schite ale arhitecturii, iar apoi cele mai bune idei pot sa fie combinate.
- **Apoi, se alege si se implementeaza un set de cazuri de utilizare, construindu-se o arhitectura imbunatatita, s.a.m.d.**
  - —In fiecare iteratie
    - Se elaboreaza elementele arhitecturale specifice aplicatiei, care pot include orice aspecte ale sistemului considerate a fi relevante d.p.d.v. arhitectural
    - Se considera fiecare caz de utilizare si se ajusteaza arhitectura pentru permite implementarea cazului de utilizare
    - Se maturizeaza arhitectura
  - —Ar trebui inceput cu *cazurile de utilizare relevante din punct de vedere arhitectural*.

Dezvoltarea unui model arhitectural

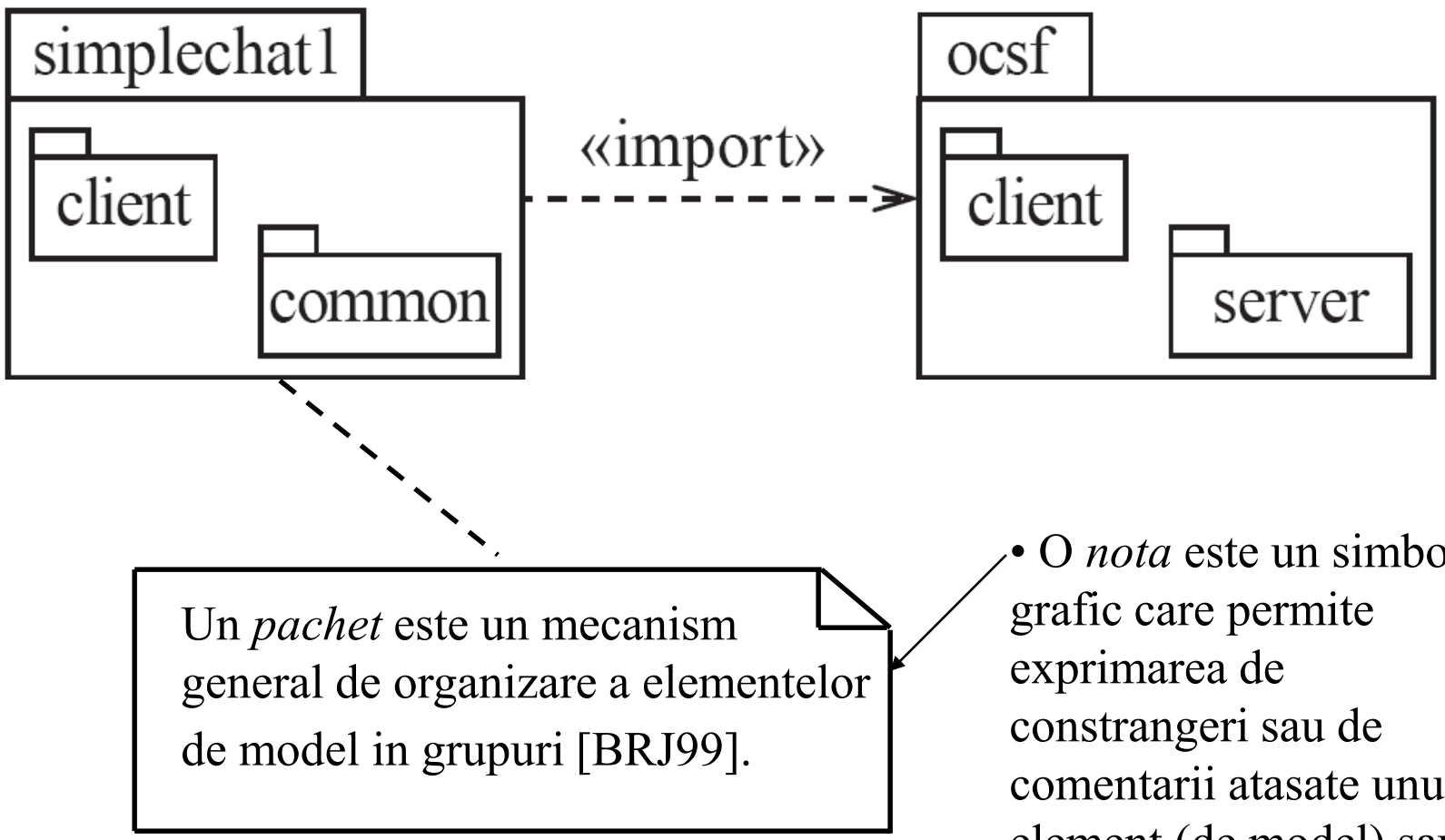
Cazurile de utilizare relevante din punct de vedere arhitectural sunt cele care [JBR99]:

- Ajuta la diminuarea celor mai semnificative riscuri tehnice sau manageriale
- Sunt cele mai importante pentru utilizatorii sistemului
  - —Exemplu
    - Se considera un sistem de tip ATM (Automated Teller Machine – automat bancar) care implementeaza patru cazuri de utilizare: Retragere Bani, Depunere Bani, Transfer Bani intre Conturi, si Vizualizare Cont.
    - Arhitectul software decide ca unicul caz de utilizare relevant din punct de vedere arhitextural este Retragere Bani [JBR99], deoarece fara acest serviciu sistemul ATM este lipsit de sens.
- Ajuta la implementarea oricarei functionalitati semnificative.

Descrierea unei arhitecturi cu UML

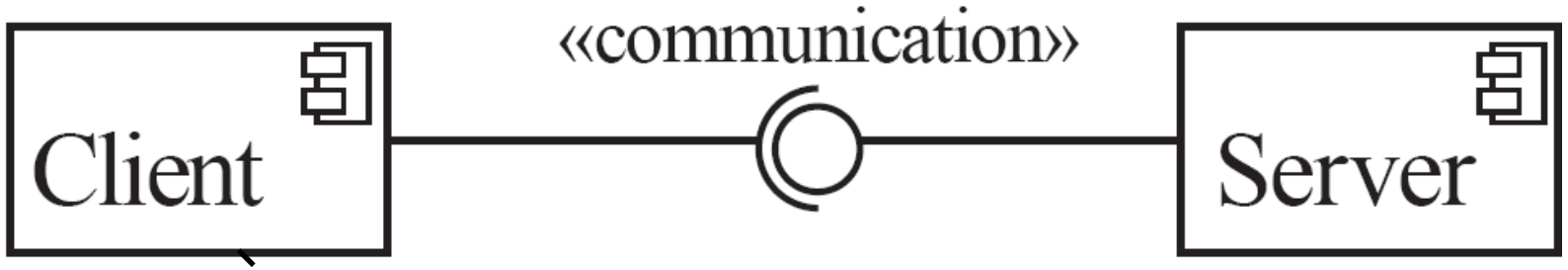
- Toate diagramele UML pot fi utile in descrierea diferitelor aspecte ale modelului arhitectural.
- Urmatoarele diagrame UML sunt in particular importante pentru modelarea arhitecturii:
  - —Diagrame de pachete
  - —Diagrame de componente
  - —Diagrame de desfasurare

Diagrame de pachete



• O *nota* este un simbol grafic care permite exprimarea de constrangeri sau de comentarii atasate unui element (de model) sau unei colectii de elemente.

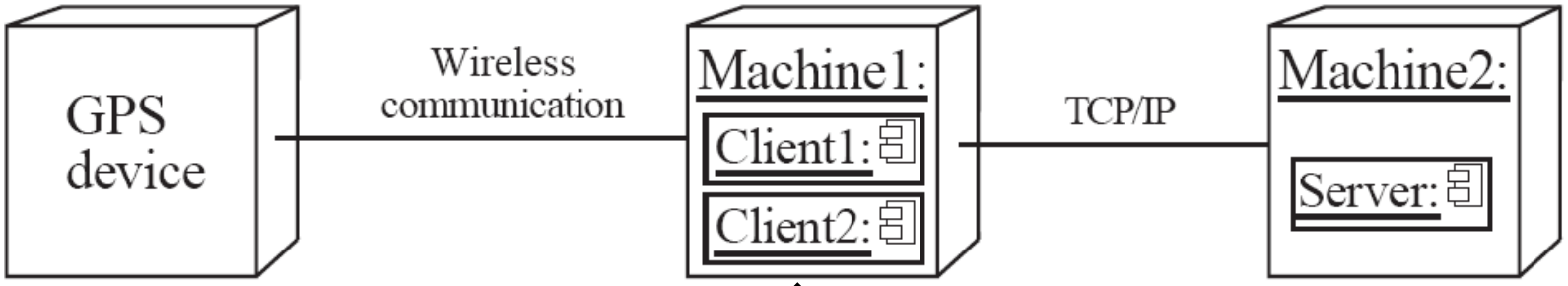
Diagrame de componente



O *componenta* este o parte fizica a unui sistem, care poate fi inlocuita cu o componenta similara si care furnizeaza realizarea (si se conformeaza) unui set de interfete [BRJ99].

- O componenta *furnizeaza* una sau mai multe interfete care pot fi utilizate de alte componente.
- Pentru vizualizarea unei componente care *utilizeaza* o interfata furnizata de o alta componenta se utilizeaza un semicerc (la capatul unei linii).

Diagrame de desfasurare



Fiecare nod dintr-o diagrama de desfasuare poate gazdui una sau mai multe componente software

Un *nod* este un element fizic care exista la executie si reprezinta o resursa de calcul (furnizand cel putin capacitate de memorare si, adesea, de procesare) [BRJ99].

9.6 Sabloane arhitecturale

• **Notiunea de *sablon* este utilizata la diferite nivele de abstractizare, inclusiv la nivelul arhitecturilor software.**

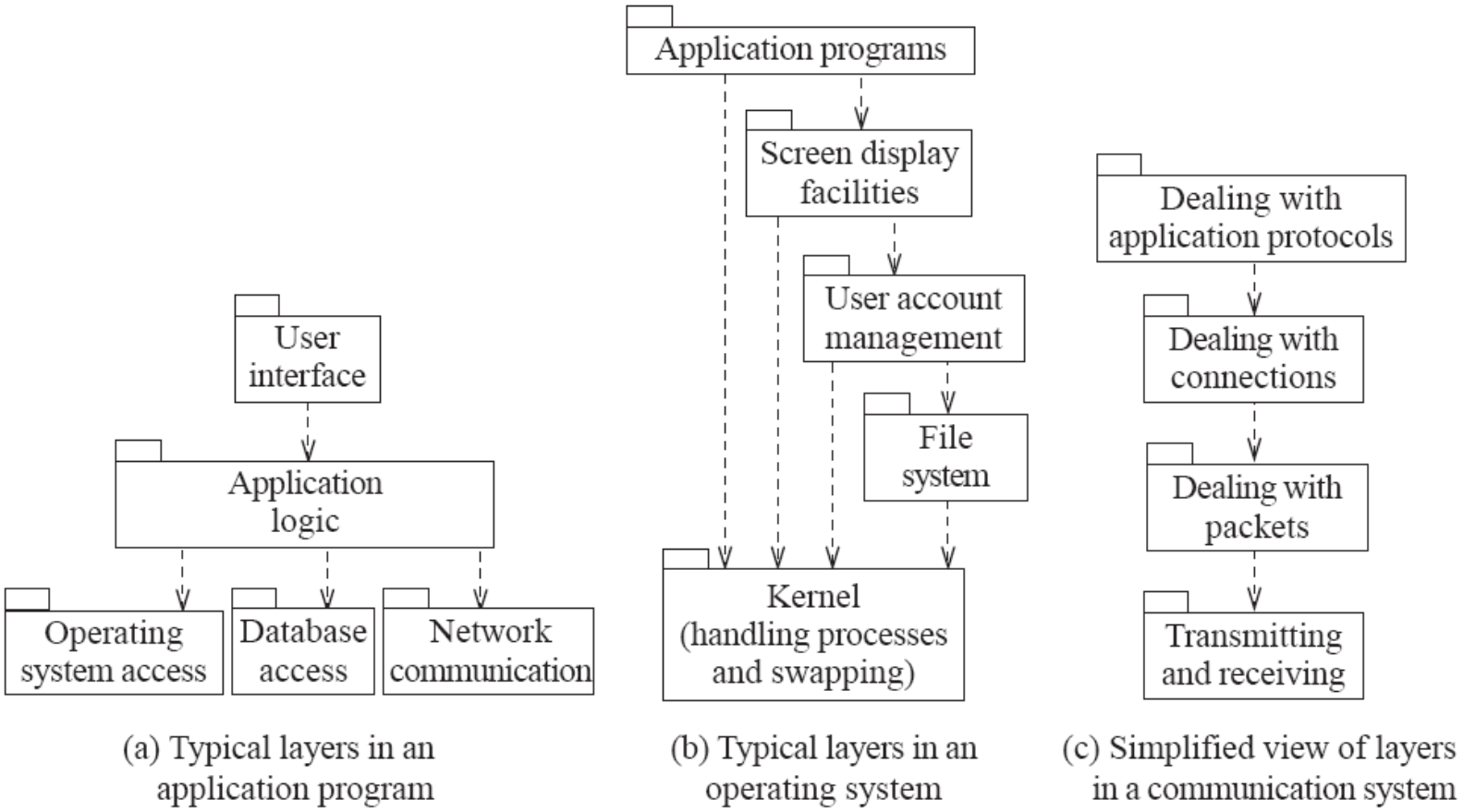
- *Sabloanele arhitecturale* (numite si *stiluri architecturale*) inlesnesc proiectarea de sisteme software alcatuite din subsisteme si componente cat mai independente cu putinta.
- Un *sablon* este schita unei solutii reutilizabile la o problema generala intalnita intr-un anumit context. Notiunea de sablon este utilizata la diferite nivele de abstractizare:
  - —*Sabloanele de proiectare* – cum sunt Observer, Façade sau Proxy – sunt larg utilizate in modelarea si proiectarea software
  - —*Sabloanele arhitecturale*– cum sunt Layers, Client-Server sau Pipe-and-Filter – reprezinta solutii standard pentru diferite probleme arhitecturale.

Sablonul arhitectural Multi-Layer

Intr-un sistem organizat pe nivele (Multi-Layer), fiecare nivel utilizeaza doar serviciile nivelelor inferioare – adesea numai serviciile nivelului imediat inferior.

- Fiecare nivel are o interfata bine definita (structura API care defineste servicile pe care le furnizeaza) utilizata de nivelele (imediat) superioare.
  - —Un nivel superior vede un nivel inferior ca un set de *servicii*.
  - —Nivelele inferioare nu cunosc detaliile sau interfetele nivelelor superioare.
- Un sistem complex poate fi construit prin suprapunerea de nivele cu grade tot mai ridicate de abstractizare.
  - —De exemplu, intr-un program de aplicatie
    - Este important sa existe un nivel separat pentru interfata utilizator (independenta acestui nivel permite aplicatiei sa functioneze cu diferite interfete utilizator)
    - Nivelele imediat inferioare interfetei utilizator furnizeaza functiile aplicatie determinate de cazurile de utilizare
    - Nivelele cele mai inferioare furnizeaza servicii pentru acces la baze de date, comunicare in retea, etc.

Exemple de sisteme Multi-Layer



(a) Typical layers in an application program

(b) Typical layers in an operating system

(c) Simplified view of layers in a communication system

Sablonul arhitectural Client-Server

- **Exista cel putin o componenta cu rol de *server*, care asteapta si apoi trateaza (sau serveste) conexiunile.**
- **Exista cel putin o componenhta cu rol de *client*, care initiaza conexiuni cu scopul de a obtine anumite servicii.**

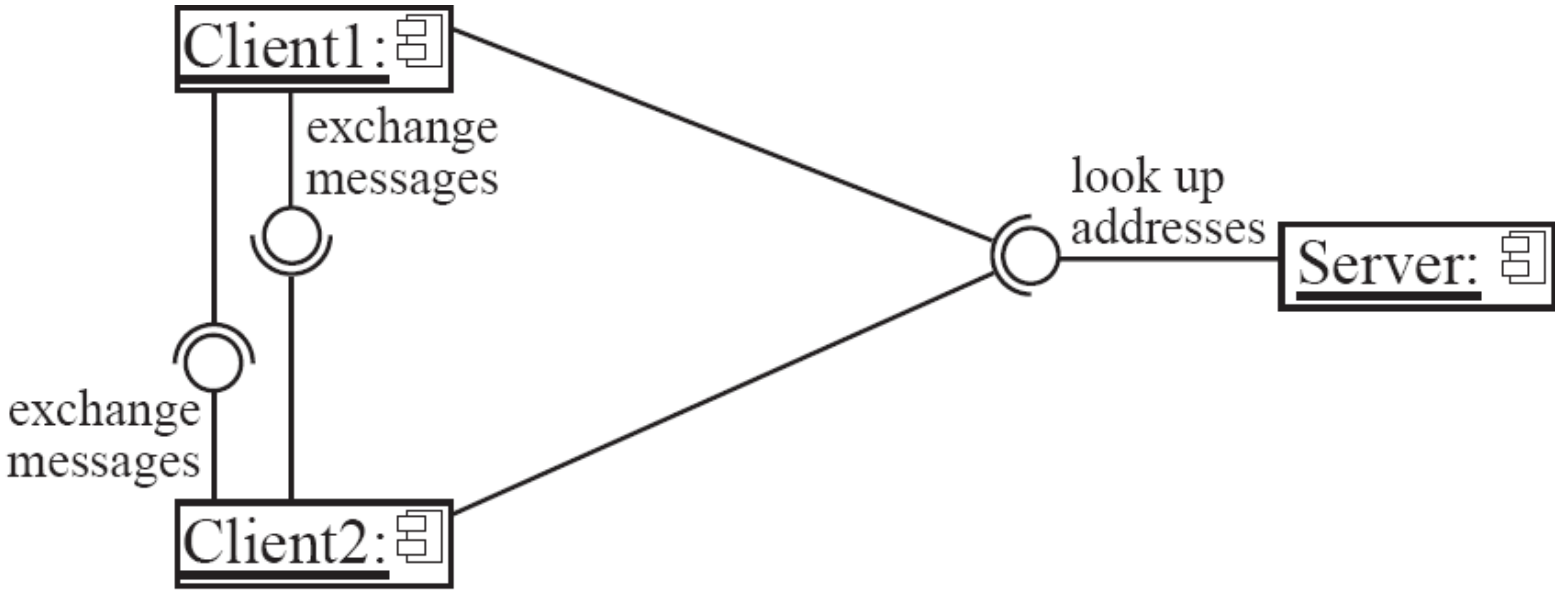
—O varianta importanta a arhitecturii (two-tier) descrisa mai sus este reprezentata de modelul **three-tier**, care include trei tipuri de noduri: clienti, servere de aplicatie si servere (de baze) de date.

» Un server de aplicatie se comporta ca si un client atunci cand acceseaza un server de date.

—O alta extensie este data de sablonul **Peer-to-Peer**. Acesta descrie o arhitectura de sistem distribuit compusa din diverse componente software.

- Fiecare dintre aceste componente poate fi atat client cat si server
- Oricare doua componente (engl. *peers*) pot stabili un canal de comunicare

O arhitectura peer-to-peer



Un server central poate fi utilizat pentru stabilirea conexiunilor intre perechi (engl. *peers*)

Sablonul arhitectural Broker

- **Sablonul arhitectural Broker inlesneste distribuirea obiectelor unui sistem software in mod *transparent* pe diferite noduri.**
  - —Utilizand arhitectura Broker, un obiect poate apela metode ale unui alt obiect fara sa stie ca acesta din urma este situat pe un alt calculator.
    - Sablonul de proiectare Proxy poate fi de folos in acest scop.
    - O componenta Broker (care trateaza toate invocarile locale de obiecte) poate fi atasata fiecarui calculator care gazduieste obiecte distribuite.
  - —CORBA (Common Object Request Broker Architecture) este un standard bine cunoscut care permite construirea acestui tip de arhitectura.

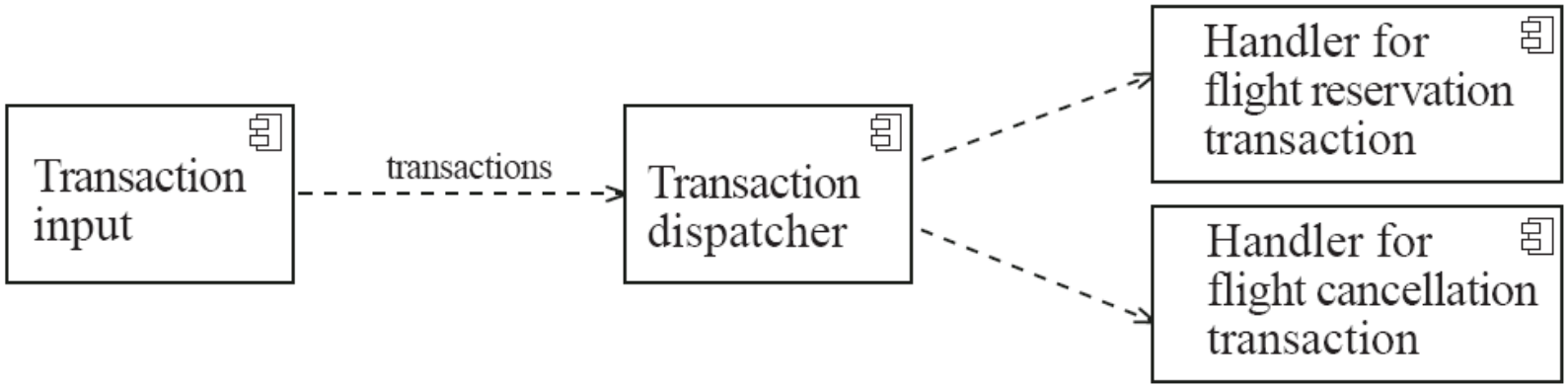
Exemplu de sistem Broker

Sablonul arhitectural Transaction Processing

In sablonul arhitectural Transaction Processing:

- Un proces citeste o serie de intrari in secventa.
  - —Fiecare intrare descrie o *tranzactie* – o comanda care in mod tipic efectueaza anumite modificari asupra datelor stocate de sistem.
- Exista o componenta *dispecer* care decide actiunea corespunzatoare fiecarei tranzactii.
- Aceasta expediaza un mesaj spre una dintr-o serie de componente care intermediaza tranzactia (engl. *transaction handler*).
  - —In fiecare caz, este esential ca tranzactia sa fie realizata complet sau deloc.

Exemplu de sistem bazat pe sablonul Transaction Processing

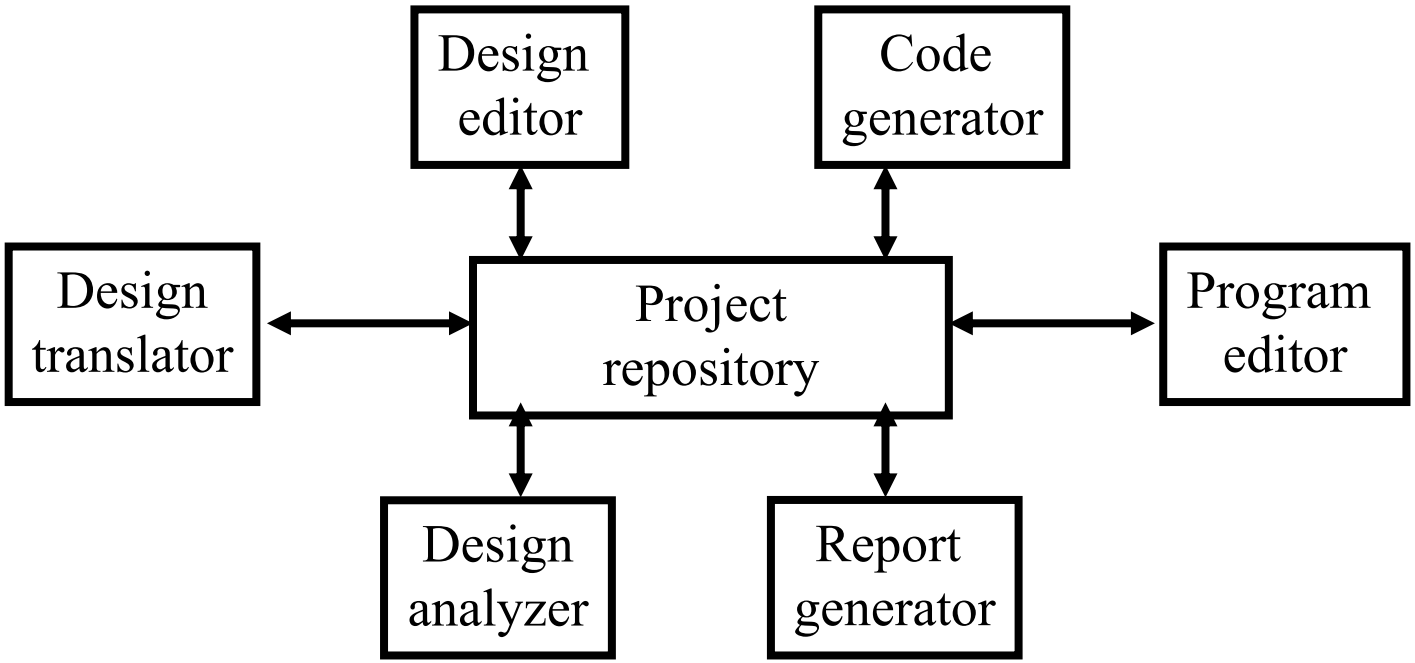


Arhitectura Transaction Processing ce poate fi utilizata in cazul sistemului de rezervari pentru curse aeriene (capitolul 5)

Sablonul arhitectural Repository

- **Sablonul Repository utilizeaza un depozit (sau baza) de date (engl. *repository*) partajat de toate subsistemele.**

  -De exemplu, acest model poate fi utilizat in proiectarea pachetelor CASE [Som04]:

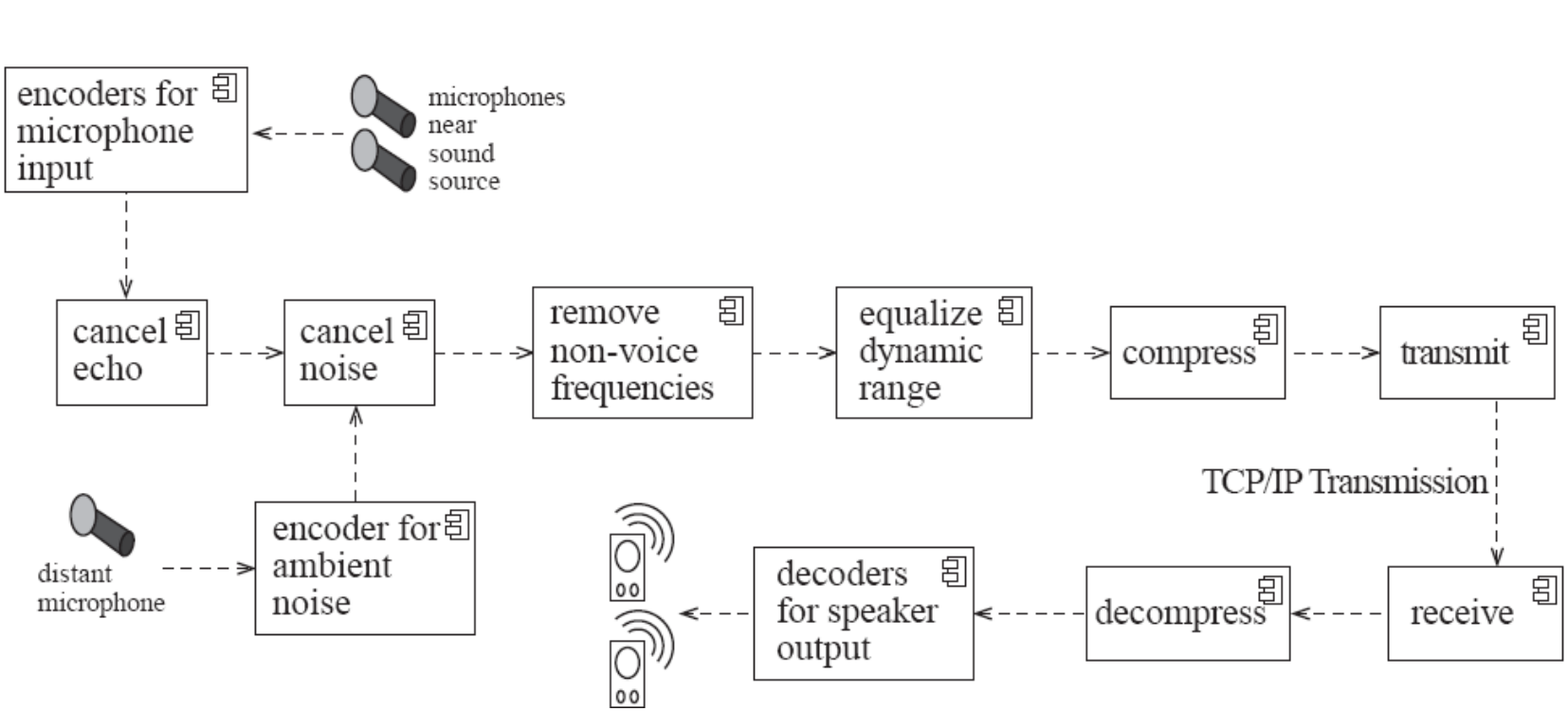


Sablonul arhitectural Pipe-and-Filter

Un flux de date, intr-un format relativ simplu, este transmis printr-o serie de procese:

- Fiecare proces transforma fluxul intr-un anumit mod.
- Fluxul de date alimenteaza arhitectura in mod continuuu.
- (Conceptual) Procesele lucreaza in mod concurent.
- Aceasta arhitectura este foarte flexibila:
  - —Aproape toate componentele ar putea fi eliminate
  - —Componentele pot fi inlocuite
  - —Se pot insera noi componente
  - —Anumite componente pot fi reordonate.

Exemplu de arhitectura Pipe-and-Filter



O arhitectura Pipe-and-Filter pentru procesare sunet

9.7 Documentul de design

Documentul de design (sau proiectare) poate sprijini inginerii software in:

- Luarea unor decizii de proiectare
- Comunicarea diverselor aspecte ale proiectului (proiectul se adreseaza celor care implementeaza, modifica sau utilizeaza sistemul)
- Revizuirea si imbunatatirea proiectului

Structura unui document de design

A. Obiective:

—Identificarea sistemului descris in documentul de design

—Referinte catre cerintele software proiectate (*traceability)*

B. Prioritati:

—Descrierea prioritatilor (de exemplu, atribute de calitate) ce ghideaza procesul de proiectare

C. Descriere de ansamblu:

—Ofera cititorului o vedere generala asupra proiectului (de exemplu, sub forma unei diagrame)

D. Probleme majore:

—Probleme majore tratate in proiect

—Solutii alternative considerate, decizii si ratiunile pe care se bazeaza

E. Detalii:

—Orice alte detalii de interes pentru cititor, care nu sunt mentionate in sectiunile precedente

In elaborarea unui document de design

- Se vor evita informatiile *evidente* pentru proiectantii sau pentru programatorii experientati.
- Se vor evita detaliile ce pot fi inserate sub forma de *comentarii* in cod.
- Se vor evita detaliile ce pot fi *extrase automat* din cod (de exemplu lista metodelor `public`).

Referinte suplimentare

[BRJ99] G. Booch, J. Rumbaugh and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[JBR99] I. Jacobson, G. Booch and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[Pre97] R. Pressman. *Software Engineering: A Practitioner's Approach*, (4th edition). McGraw-Hill, 1997.

[SG96] M. Shaw and D. Garlan. *Software Architecture*. Prentice-Hall, 1996.

[Som04] I. Sommerville. *Software Engineering,* (7th edition). Addison-Wesley, 2004.