Principii de proiectare orientate pe obiect

Proiectarea orientată pe obiecte (OOD : Object Oriented Design) este o metodă de descompunere a arhitecturii unui sistem software cu scopul obţinerii modularizării acestuia.

OOD se bazează pe programarea orientată pe obiect, implicit pe obiectele pe care orice sistem sau subsistem le manipulează.

Se poate spune că OOD este relativ independent faţă de limbajul de programare folosit (Java, C++).

Pentru construirea unui design bun al sistemelor software, trebuie respectate mai multe principii de baza ale OOD, respectarea acestora putând fi privită ca o modalitate de rezolvare a acestor probleme.

5.1. Principiul Deschis – Închis - OCP (Open Close Principle)

Principiul Deschis – Închis, a fost formulat de către Bertrand Meyer, astfel:

"*Orice entitate software (clase, module, funcţii, etc) ar trebui sa fie **deschisă** pentru **extindere**, şi **închisă** pentru **modificare**."*

Prin utilizarea principiului deschis – închis se evită fragilitatea, rigiditatea si imobilitatea piesei de soft, proiectându-se module care "să nu se modifice niciodată".

În proiectarea sistemului de foloseşte **abstractizarea şi polimorfismul**.

Când specificaţiile sistemului se modifică, comportamentul modulelor se extinde prin adăugarea de cod nou, fără a interveni cu modificări în codul deja existent, care funcţionează.

Un modul software care respectă principiul deschis – închis are două caracteristici principale:

1. "**deschis pentru extindere**": comportamentul modulului poate fi extins. Se poate obţine astfel un comportament nou în conformitate cu noile cerinţe ale aplicaţiei.
2. **"închis pentru modificări"**: codul sursă al modulului este "inviolabil", nimănui nu i se permite să facă schimbări în cod.

Deşi la o prima vedere cele două cerinţe par contradictorii, totuşi ele pot fi simultan satisfăcute prin utilizarea **abstractizării**.

În limbajele de programare orientate pe obiecte, se pot crea abstractizări care sunt fixe din punct de vedere al codului, dar care totuşi reprezintă un grup nelimitat de posibile comportamente.

Abstractizările sunt clasele de bază abstracte, iar grupul nelimitat de comportamente este dat de toate clasele ce se pot deriva din acestea.

Deci, prin utilizarea abstractizării se îndeplineşte restricţia impusă de OCP: modulele sunt scrise astfel încât să poată fi extinse fără a fi modificate.

În **Fig. 5.1** se prezintă un exemplu foarte utilizat de proiectare a unei aplicaţii simple, în care atât clasa *Client* cât şi clasa *Server* sunt două clase concrete.

Clasa *Client* utilizează clasa *Server*.

Dacă se doreşte ca un obiect *Client* să utilizeze un alt obiect *Server*, atunci clasa *Client* trebuie modificată pentru a indica numele noii clase server, deci aceasta proiectare **nu respectă OCP**.

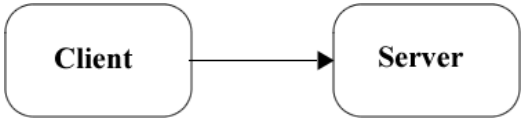


Fig. 5.1 Proiectare greşită Client-Server

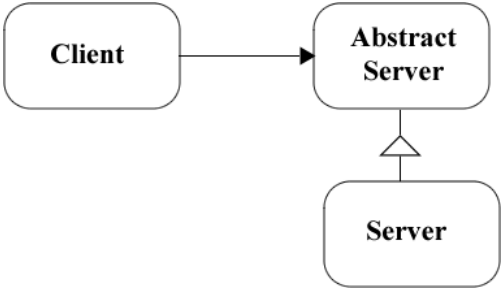


Fig. 5.2. Proiectare corectă Client-Server

Fig. 5.2 prezintă design-ul pentru aplicaţia **Server – Client** astfel încât să respecte principiul deschis – închis.

În acest caz, se utilizează o clasă abstractă *AbstractServer* , din care se derivează clasa concreta *Server* şi clasa *Client* depinde de această clasa abstractă.

Obiectele *Client* vor folosi obiecte ale unei clasei derivate *ServerA*.

Dacă se doreşte ca obiectele *Client* să folosească obiecte ale unei alte clase *Server*, atunci se creează, prin derivare, din clasa abstractă o nouă clasă concretă *ServerB*.

În acest mod, codul clasei *Client* rămâne nemodificat.

Pentru obţinerea unor noi comportamente necesare extinderii aplicaţiei, trebuie doar să se deriveze din clasa abstractă, clase concrete *Server* care implementează noi comportamente.

Închidere strategică

“Nici un program nu poate fi închis 100%.” .

Nici un program nu poate fi închis pentru modificări în mod total, deoarece, oricât de închis ar fi un modul, întotdeauna pot apărea situaţii pentru care nu a fost închis.

Având în vedere că, închiderea unui modul faţă de modificări nu poate fi completă, ea trebuie să fie **strategică**.

Închiderea strategică presupune ca designer-ul arhitecturii sistemului să abstractizeze partea care este cea mai susceptibilă a fi extinsă.

Închiderea explicită a sistemului la modificări se obţine utilizând **abstractizarea**.

Clasa abstractă furnizând metode care pot fi invocate dinamic şi în cadrul cărora se stabilesc politicile de decizie la nivel general.

O altă metodă care ajută la închiderea sistemului se bazează pe abordare “data-driven”, care presupune plasarea codului care se referă la deciziile de politică volatilă într-o locaţie separată, fie într-un alt fişier, fie într-un alt obiect, astfel că pe viitor modificările se vor face într-un număr minim de locaţii.

Reguli de proiectare folosite în OOD

Principiul deschis – închis este sursa unor reguli de proiectare folosite în OOD. Acestea se referă la variabile private şi variabilele globale folosite în program:

Toate variabilele să fie private

Aceasta este dintre cele mai întâlnite convenţii în OOD.

Variabilele unei clase trebuie să fie cunoscute doar în metodele definite în clasa respectivă.

Aceste variabile nu trebuie cunoscute de către alte clase, nici măcar de clasele derivate.

De aceea, este recomandat să fie declarate *private*, şi nu *public* sau *protected*.

- când variabilele dintr-o clasă se modifică, fiecare dintre clasele care depind de acestea ar trebui modificate.

Deci, nici o funcţie care depinde de o variabilă, nu poate fi închisă faţă de aceasta.

Datele constante (declarate *const* în C++ sau *final* în Java) pot fi publice sau protejate, fără a încălca principiul închiderii.

În OOD, închiderea celorlalte clase, inclusiv a claselor derivate, faţă de modificarea variabilelor dintr-o clasă, poartă numele de **încapsulare datelor.**

Avantaje ale încapsulării datelor:

- **securitatea datelor**: acestea nu pot fi modificate din exteriorul clasei în care sunt vizibile;
- **consistenţă**: prin modificarea variabilelor din alte clase, de către diferiţi utilizatori pot apărea situaţii de inconsistenţă, datorate unor modificări care nu sunt atomice.

Fără variabile globale

Argumentele împotriva variabilelor globale sunt similare celor împotriva variabilelor publice.

Nici un modul care utilizează o variabilă globală nu poate fi închis faţă de celelalte module care modifică respectiva variabilă.

Este posibil ca un modul să utilizeze variabila globală într-un mod neaşteptat pentru celelalte module care o mai folosesc.

Designerul trebuie să evalueze cât din închiderea aplicaţie este "sacrificată" în favoarea variabilelor globale şi să determine dacă avantajele oferite de utilizarea variabilelor globale compensează dezavantajele date de utilizarea lor.

Concluzii

În multe privinţe acest principiu este considerat esenţa proiectării orientate pe obiecte.

Avantajele care se obţin de pe urma folosirii acestuia sunt reutilizarea codului şi mentenabilitatea software-ului.

Un design bine gândit poate fi extins fără modificări, noile caracteristici ale sistemului adăugându-se prin completarea de cod nou, şi nu prin modificarea celui existent.

Mecanismele pe care se sprijină acest principiu sunt ***abstractizarea şi polimorfismul***.

Faptul ca se programează într-un limbaj orientat pe obiecte nu duce automat la concordanţă / conformitate cu OCP, ci este necesar ca designerul să aplice abstractizarea pe acele părţi ale programului care sunt susceptibile de a fi modificate.

Crearea abstractizărilor şi apoi derivarea claselor concrete din aceste abstractizări, duc la obţinerea unor module deschise pentru extindere şi închise pentru modificare.

Mecanismul care asigura crearea acestor clase derivate este moştenirea, iar principiul substituţiei al lui Liskov este cel care ghidează designul acestor ierarhii de clase.

5.2. Principiul substituţiei Liskov

Principiul substituţiei al lui Liskov (Liskov Substitution Principle, LSP) a fost enunţat în literatura de specialitate astfel:

Funcţiile care utilizează pointeri sau referinţe către clasele de bază, trebuie să poată utiliza obiecte ale claselor derivate din clasa de bază în mod transparent [Riel 1996].

Doar atunci când obiectele claselor derivate pot înlocui complet obiecte ale claselor de bază, codul poate fi reutilizat, atingându-se astfel scopul OCP (obţinerea unui cod reutilizabil, prin extindere

şi nu prin modificare); deci Principiul substituţiei al lui Liskov poate fi interpretat ca un mijloc de verificare a codului, din punctul de vedere al obţinerii unui design în acord cu OCP.

Acest principiu creează ierarhii de clase care se conformează OCP.

Să presupunem că există o metodă care nu se conformează LSP, atunci acea metodă utilizează un pointer sau o referinţă către clasa de bază, şi în acelaşi timp "ştie", conform presupunerii de mai sus, despre toate clasele derivate din clasa de bază.

O astfel de metodă încalcă OCP deoarece trebuie modificată ori de cate ori o nouă clasă derivată din clasa de bază este creată.

Exemple pentru ilustrarea LSP: Fie o clasă *Dreptunghi*:

```
public class Dreptunghi {
private double  width;
private double  height;
public double getHeight() {
  return height;
}
public void setHeight(double height) {
  this.height = height;
}
```

```
public double getWidth() {
  return width;}
public void setWidth(double width) {
  this.width = width;}}
```

Atât în C++ cât şi în Java, moştenirea se bazează pe modelul relaţional.

Modelul relaţional ISA descrie o relaţie strictă între clase, în care membrii unei clase formează o submulţime a unei alte clase.

Modelul ISA (“is a”) pune în evidenţă că un obiect (*RombA* din Fig. 5.3.) care aparţine unei subclase (clasa *Romb* din Fig. 5.3), aparţine simultan tuturor super-claselor (deci *RombA* este o instanţă a lui *Romb*, dar este în acelaşi timp instanţă şi a claselor *Patrulater* si *Poligon*)

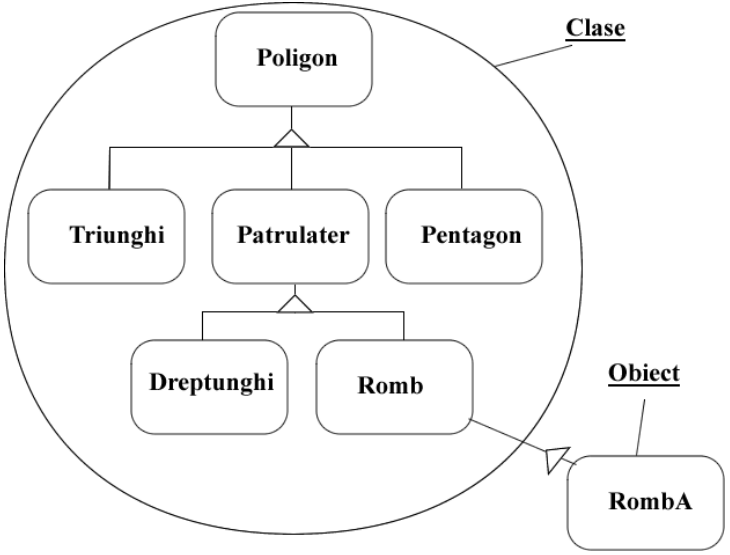


Fig. 5.3. Modelul relaţional ISA

Utilizarea modelului ISA este considerată ca fiind o tehnică de bază în Analiza Orientată pe Obiecte (Object Oriented Analysis, OOA).

În continuarea exemplului de mai sus, să presupunem că la un moment dat, în decursul dezvoltării unor noi aplicaţii, proiectantul are nevoie să utilizeze şi pătrate.

Deoarece un pătrat este un dreptunghi cu toate laturile egale, putem modela clasa *Pătrat* prin derivare din clasa *Dreptunghi*, în conformitate cu modelul ISA relationship.

Dar acest mod de a gândi, poate duce la anumite probleme pe care le vom întâmpina când trecem efectiv la scrierea codului.

Prima problemă, şi cea mai importantă, este legată de variabilele *heigth* şi *width*; pentru a defini un pătrat nu avem nevoie şi de lăţime si de înălţime (deci de amândouă), dar clasa *Pătrat* le va moşteni pe amândouă.

O problemă secundară este legată de folosirea eficientă a memoriei, mai ales dacă se utilizează sute de obiecte ale clasei *Pătrat*.

Dar trecând peste acest aspect al eficienţei memoriei utilizate, o problemă importantă este cea generată de existenţă celor două metode *setWidth* şi *setHeight*.

În primul rând, aceste metode sunt inadecvate pentru clasa *Pătrat*, deoarece dimensiunile unui pătrat sunt egale.

Iată deci o problemă de design, care totuşi poate fi depăşită dacă modificăm codul celor două metode astfel: când se setează "lăţimea" unui obiect *Pătrat*, "înălţimea" va fi ajustată în mod corespunzător, şi reciproc.

În acest mod un obiect *Pătrat* îşi păstrează proprietăţile matematice.

```
public class Pătrat extends Dreptunghi{
 public void setWidth(double width) {
      super.setWidth(width);
      super.setHeight(width);
 }
 public void setHeight(double height) {
      super.setHeight(height);
      super.setWidth(height);
 }     }
```

Dar dacă pentru a deriva, trebuie modificată clasa de bază, înseamnă ca aceasta are o problemă de proiectare.

Mai mult, acest lucru reprezintă o încălcare a OCP, deoarece programul ar trebui sa fie închis pentru modificări.

Consistenţa unui model

În acest moment, există două clase, care în aparenţă funcţionează corespunzător: *Pătrat* şi *Dreptunghi,* fiecare modelează corespunzător obiectele matematice ale căror nume le poartă (pătrat şi dreptunghi).

În aparenţă, se poate spune că sistemul are un comportament consistent.

Se poate trage concluzia că modelul este auto-consistent şi corect.

Dar această concluzie ar fi o eroare **deoarece un model care este auto-consistent nu este în mod necesar consistent şi cu toţi utilizatorii săi.**

Să considerăm următoarea funcţie:

```
void g(Dreptunghi r)
 {
 r.setWidth(5);
 r.setHeight(4);
 assert ((r.getWidth() * r.getHeight()) == 20) ;
 }
```

În mod evident aceasta metodă funcţionează corespunzător pentru *Dreptunghi*, dar pentru *Pătrat* generează o aserţiune falsă.

Aceasta este o problemă gravă.

Desigur, programatorul care a scris această metodă a făcut o presupunere îndreptăţită şi anume că modificarea lăţimii unui dreptunghi lasă înălţimea acestuia neschimbată.

Funcţia *g(Dreptunghi)* este un exemplu de funcţie care acceptă referinţe către *Dreptunghi* dar care nu funcţionează corespunzător pentru obiectele din clasa *Pătrat*.

Acest tip de funcţii încalcă principiul substituţiei al lui Liskov.

Validitatea nu este intrinsecă

Cazul expus mai sus impune o concluzie importantă: *validitatea unui model nu poate fi considerată semnificativă decât într-un anumit* ***context***.

Un model izolat nu poate fi apreciat din punct de vedere al validităţii lui.

Validitatea unui model se exprimă în termenii clienţilor săi.

Spre exemplu, în cazul de mai sus când s-a examinat versiunea finală a claselor *Pătrat* şi *Dreptunghi*, în mod izolat, s-a ajuns la concluzia că sunt consistente şi valide.

Totuşi, examinându-le din punctul de vedere al unui programator care a făcut respectiva presupunere în legătură cu clasa de bază, s-a ajuns la concluzia că modelul este greşit.

Deci, pentru aprecierea unui proiect, analiza trebuie să fie făcută într-un context şi nu în mod izolat.

Design-ul trebuie văzut din perspectiva utilizatorilor acestuia; ei lucrează pe baza unor presupuneri rezonabile asupra modului de funcţionare a modelului.

Trebuie analizat de ce modelul claselor *Pătrat* şi *Dreptunghi,* care părea corect, nu a funcţionat.

Nu este *pătratul* un *dreptunghi*? Nu a funcţionat relaţia modelului ISA, orice obiect al clasei *Pătrat* aparţine sau nu şi clasei *Dreptunghi*?

Un pătrat poate fi un dreptunghi, dar din punct de vedere strict matematic.

Un pătrat ca obiect al clasei *Pătrat* nu este un obiect *Dreptunghi*, deoarece comportamentul unui obiect *Pătrat* nu este consistent cu comportamentul unui obiect *Dreptunghi*, iar clasele tocmai acest aspect trebuie sa-l modeleze: **comportamentul**.

Prin comportament se înţelege **comportamentul public**, extrinsec, cel pe care se bazează clienţii, şi nu comportamentul privat, intrinsec al obiectului.

Spre exemplu, autorul funcţiei *g(),* de mai sus, s-a bazat pe faptul că pentru un obiect al clasei *Dreptunghi*, lăţimea şi înălţimea variază independent. Această independenţă a celor două variabile este un comportament public extrinsec pe care, probabil, contează şi alţi programatori.

toate clasele derivate trebuie să respecte comportamentul pe care clienţii îl aşteaptă de la clasa de bază pe care o utilizează.

Design prin contract

Există o strânsă relaţie între Principiul substituţiei al lui Liskov şi conceptul de “Design by contract”, aşa cum a fost definit de Bertrand Meyer.

Utilizând această schemă, metodele claselor declară precondiţii şi postcondiţii.

O metodă se execută dacă precondiţiile sunt adevărate.

La terminarea metodei, aceasta garantează că postcondiţiile vor fi adevărate.

Aplicând pe exemplul de mai sus, putem spune că postcondiţia metodei s*etWidth(double w)* în clasa *Dreptunghi* este:

```
assert((width == w) && (height == old.height))
```

Regula care se aplică precondiţiilor şi postcondiţiilor la derivare, aşa cum a formulat-o B. Meyer este următoarea:

Când se redefineşte o metodă în clasa derivată, precondiţia se înlocuieşte prin una mai "slabă", iar postcondiţia prin una mai "puternică".

Cu alte cuvinte, utilizarea unui obiect se face prin intermediul interfeţei clasei de bază şi utilizatorul cunoaşte doar precondiţiile şi postcondiţiile acestei clase.

Deci, obiectele claselor derivate nu trebuie să impună precondiţii mai puternice decât ale clasei de bază, ele trebuie să accepte orice precondiţie pe care clasa de bază o acceptă.

De asemenea, clasele derivate trebuie să se conformeze tuturor postcondiţiilor clasei de bază, comportamentul şi ieşirile lor nu trebuie să încalce nici una din constrângerile impuse de superclasă.

Postcondiţia din metoda *setWidth(double w)* din clasa *Pătrat* este mai puţin restrictivă decât cea din metoda s*etWidth(double w)* din clasa *Dreptunghi*, deoarece nu se conformează celei din superclasa, mai exact nu respectă şi condiţia: (height == old.height*)*.

Deci, metoda *setWidth(double w)* din clasa *Pătrat* încalcă contractul clasei de bază.

Concluzii

Respectarea OCP are ca efect obţinerea unor aplicaţii mai robuste, cu o mai bună mentenabilitate şi reutilizabilitate a codului.

Principiul substituţiei este o caracteristică importantă a acelor programe/aplicaţii care respectă OCP, deoarece tipurile derivate pot înlocui tipurile de bază astfel încât codul claselor de bază poate fi oricând reutilizat şi codul claselor derivate oricând modificat.

Substituţia claselor de bază cu obiecte ale claselor derivate este posibilă, deoarece clasele derivate respectă comportamentul extrinsec al superclaselor.

Obiectele subclaselor, conform LSP, se comportă într-o manieră consistentă cu “promisiunile” făcute de superclasă în API.

Astfel, claselor client li se furnizează un comportament stabil, pe care se pot baza.

5.3. Principiul Inversării Dependenţelor - DIP

Structura unui program, rezultată în urma respectării OCP şi LSP, este generalizată în principiul inversării dependenţelor (The Dependency Inversion Principle, DIP), formulat astfel:

A. Modulele de pe nivelele superioare nu trebuie să depindă de modulele de pe nivelele inferioare. Modulele de pe cele două nivele ar trebuie să depindă de nişte abstractizări.

B. Abstractizările nu trebuie să depindă de detalii. Detaliile trebuie să depindă de abstractizări.

Acest principiu este mecanismul de realizare a Principiului Deschis-Închis (OCP), deoarece prin aplicarea DIP se creează o arhitectură a sistemului închisă pentru modificări (abstractizările de pe nivelul superior) şi deschisă pentru extindere (clase concrete de pe nivelul inferior), deci modulele sunt reutilizabile şi stabile.

Metodele tradiţionale de dezvoltare a software-ului (programarea structurală) creează structuri în care modulele de pe nivelele superioare depind de cele de pe nivelele inferioare şi abstractizările depind de detalii de implementare.

Normal este ca nivelul superior să impună schimbări în nivelul inferior, modulele de pe nivelul superior să fie independente faţă de cele de pe nivelul inferior.

Astfel, **DIP impune** ca, pe de o parte, **într-o ierarhie de clase, clasa din care se derivează să nu cunoască nici una dintre subclasele sale**, iar pe de alta parte, **modulele care implementează detaliile depind de abstractizări, şi nu invers**.

Fie exemplul din **Fig. 5.4** al unui program alcătuit din trei module.

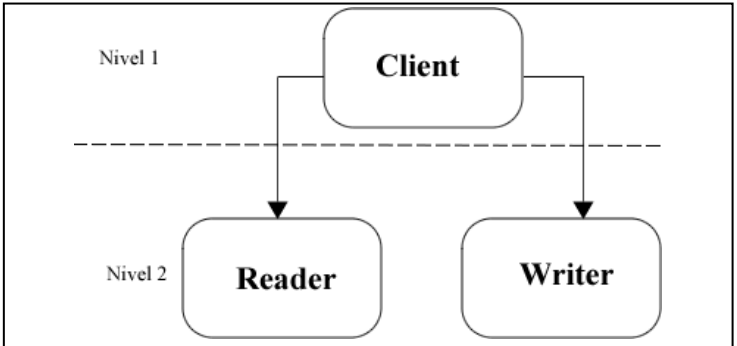


Fig. 5.4. Structura unui program alcătuit din trei module

Modul de pe nivelul 1, *Client*, implementează logica programului, iar modulele de pe nivelul 2 realizează anumite operaţi, conform deciziilor luate de modulul *Client*.

Aşa cum se observă şi din figură, modulul *Client* este dependent de modulele de pe nivelul 2.

Dacă modulele de pe nivelul 2 mai pot fi reutilizate în aplicaţii în care să îndeplinească aceleaşi funcţii, modulul de pe nivelul 1 este nereutilizabil el fiind strict dependent de modulele de pe nivelul inferior, cel mult poate fi reutilizat într-un context care să implice celelalte două module.

Dacă modulul *Client* poate fi modificat astfel încât să devină independent de celelalte module, atunci el poate fi reutilizat fără probleme, în alte programe de aceeaşi natură.

OOD ne pune la dispoziţie un mecanism pentru a face acest lucru: **inversarea dependenţelor**.

Dacă sistemului din Fig. 5.4 i se aplică principiul inversării dependenţelor, conform căruia modulele de pe nivele superioare nu depind de modulele de pe nivelele inferioare ci de nişte abstractizări, iar abstractizările nu depind de “detalii”, se obţine diagrama de clase din Figura 5.5.

Astfel, în această proiectare apar două clase abstracte "AbstractReader" şi "AbstractWriter". Modulul *Client* depinde de aceste două abstractizări. În acest fel a fost înlăturată dependenţa clasei Client faţă de clasele Reader şi Writer.

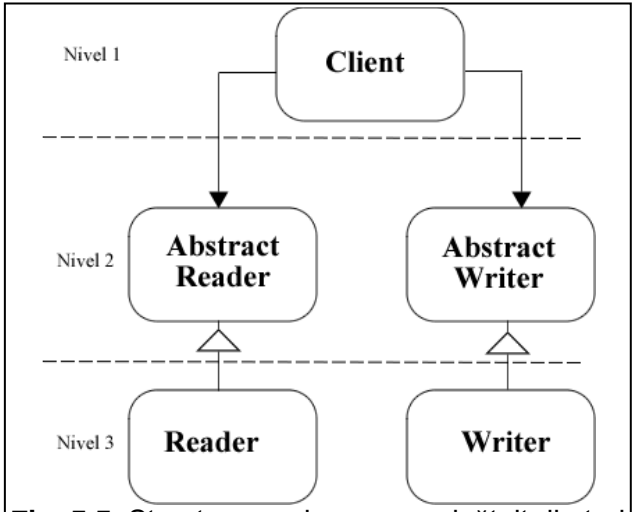


Fig. 5.5. Structura unui program alcătuit din trei module în urma aplicării DIP

Dependenţa a fost inversată, în sensul că atât clasa Client cât şi clasele Reader şi Writer depind de cele două clase abstracte.

Cu acest nou design, clasa Client poate fi reutilizată şi în alte contexte, independent de clasele concrete Reader şi Writer.

Se pot adăuga noi clase derivându-se din clasele abstracte AbstractReader, respectiv AbstractWriter; clasa Client nu va depinde de nici una din clasele nou create prin derivare.

În acest fel clasa Client a devenit **mobilă**.

În general, prin utilizarea principiul inversării dependenţelor se obţine o structură organizată pe nivele, cu modulele reutilizabile pe cele două nivele.

Modulele de pe nivelul inferior sunt reutilizate în forma librăriilor.

Organizarea pe nivele

Conform lui Booch [Booch, 1996]:

"[…] Toate arhitecturile orientate pe obiect, bine structurate au nivele clar definite, fiecare nivel oferind un set de servicii prin intermediul interfeţei sale."

O interpretare simplistă a acestei afirmaţii ar putea duce la realizarea unei arhitecturi în care dependenţele ar fi “pasate” de la un nivel la altul, având în vedere că **dependenţa este tranzitivă**.(**Fig. 5.6**).

Nivelul Deciziilor depinde de **Nivelul Mecanismelor**, care depinde de **Nivelul Serviciilor**, deci **Nivelul Deciziilor** depinde de **Nivelul Serviciilor** (**tranzitivitatea dependenţelor**).

O structură corectă este cea în care nivelele inferioare oferă servicii prin intermediul interfeţelor abstracte. (**Figura 5.7**).

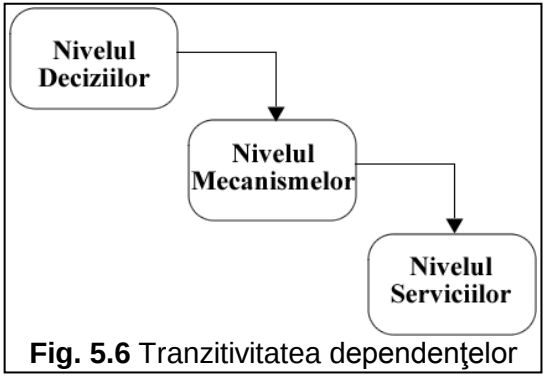


Fig. 5.6 Tranzitivitatea dependenţelor

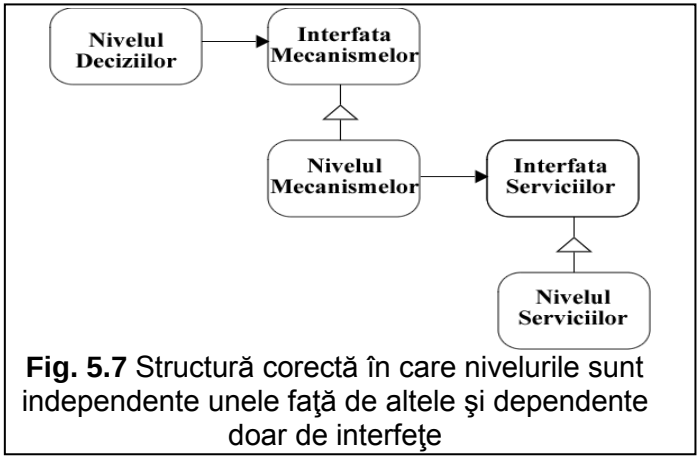


Fig. 5.7 Structură corectă în care nivelurile sunt independente unele faţă de altele şi dependente doar de interfeţe

Interfaţa reprezintă *totalitatea serviciilor pe care nivelul inferior le oferă nivelului superior.*

Deci, toate nivelele sunt independente unele faţă altele, şi dependente doar de clasele abstracte.

Cu alte cuvinte, conform acestui model, **Nivelul Deciziilor** este complet independent de nivelele inferioare, putând fi reutilizat în orice alt context în care se defineşte un nivel inferior bazat pe interfaţa **Nivelului Mecanismelor**.

Deci, inversând dependenţele, se creează o structură care are toate calităţile unui bun design: flexibilitate, durabilitate şi mobilitate.

Comparaţie între o arhitectura procedurală şi una OO

Comparaţia între o arhitectură procedurală şi una orientată-obiect are ca scop să pună în evidenţă avantajele şi dezavantajele fiecăreia.

În **Fig. 5.8** este schiţată arhitectura unui sistem procedural, în **Fig. 5.9** este schiţată arhitectura unui sistem orientat pe obiecte.

Dezavantajele unei arhitecturi procedurale sunt:

- nu există o separare clară a nivelului decizional al aplicaţiei de nivelul mecanismelor de implementare şi de cel al detaliilor;
- orice modificare în modulele de pe nivelul inferior duce la modificări în modulele de pe nivelul superior.

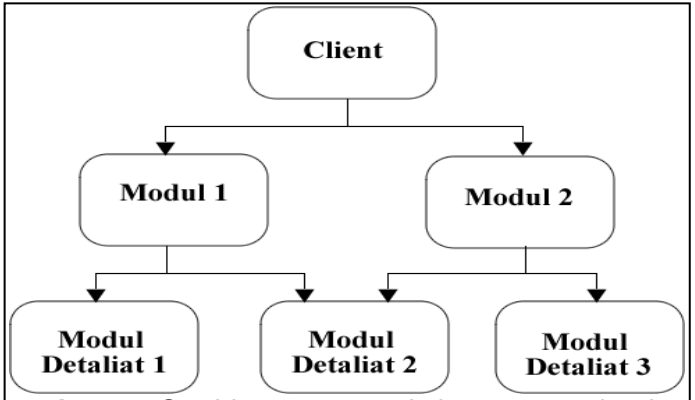


Fig. 5.8. O arhitectură a unui sistem procedural

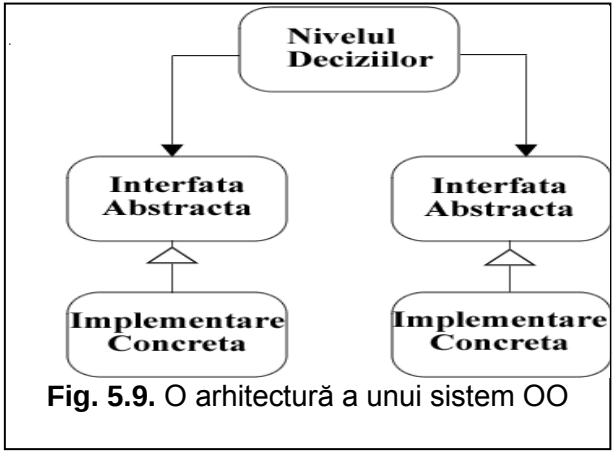


Fig. 5.9. O arhitectură a unui sistem OO

Aceste dezavantaje sunt înlăturate în cadrul unei arhitecturi OO, care respectă principiile OCP, LSP şi DIP:

- prin organizarea pe nivele se separă clar nivelul decizional de cel al implementării detaliilor,
- prin inversarea dependenţelor, obţinându-se o arhitectură alcătuită din piese de software flexibile, mobile şi uşor de reutilizat, deci toate atributele unui bun design.

După cum se observă, într-o arhitectură OO, modulele care au detalii de implementare nu depind de un alt modul, ci numai de abstractizări.

Reguli de proiectare folosite în OOD

Prin utilizarea Principiului Inversării Dependenţelor, s-a ajuns la câteva rezultate practice.

Utilizarea interfeţelor (claselor abstracte) pentru a evita legăturile directe între clasele concrete ***(Fig.5.10)***

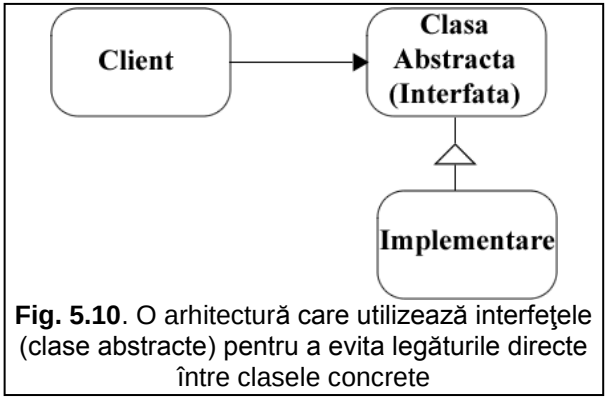


Fig. 5.10. O arhitectură care utilizează interfeţele (clase abstracte) pentru a evita legăturile directe între clasele concrete

Utilizarea interfeţelor contribuie la obţinerea unui **cod stabil** pentru clasa concretă Client.

O clasa abstractă este mai puţin probabil să fie modificată, pe de altă parte o clasă abstractă este mai uşor de extins/modificat.

Totuşi, trebuie avut în vedere că a modifica o abstractizare contravine OCP, care se bazează pe stabilitatea abstracţiunii.

Regulă de bază - Evitarea dependenţelor tranzitive prin utilizarea interfeţelor!

Concluzii

Principiului Inversării Dependenţelor este sursa multor beneficii ale tehnologiei orientate pe obiect; prin aplicarea lui se obţin module reutilizabile.

Este foarte important pentru construcţia codului, ca acesta să fie rezistent, robust şi elastic la modificări.

Deoarece abstractizările şi detaliile aplicaţiei sunt izolate unele de altele, codul este mult mai uşor de întreţinut (mentenabilitate crescută).

Cheia din principiul inversării dependenţelor este utilizarea abstractizării.

Detaliile de implementare sunt izolate unele de altele, între ele fiind interpuse abstractizările (care sunt clase stabile), astfel o modificare efectuată într-un modul ce implementează detalii nu se propagă în întreg sistemul.

Izolarea claselor care implementează detalii şi dependenţa acestora doar de abstractizări contribuie la obţinerea unor clase care pot fi utilizate, cu uşurinţă, în alte aplicaţii.

Privit din perspectiva principiilor prezentate în secţiunile anterioare, se poate spune că OCP este **scopul**, DIP (Principiului Inversării Dependenţelor) asigură **mecanismul** de îndeplinire a scopului, iar LSP (Principiul substituţiei al lui Liskov) este modalitatea de **verificare** a mecanismului.

Care este scopul unui design conform OCP ?

Obţinerea unor entităţi software care să fie deschise pentru extindere dar închise la modificări, în acest timp păstrându-se calităţile unui bun design: flexibilitate, mobilitate şi reutilizabilitate.

DIP specifică modalitatea de atingerea acestui scop:

- într-o ierarhie de clase, clasa din care se derivează nu cunoaşte nici una dintre subclasele sale
- modulele care implementează detaliile depind de abstractizări, şi nu invers.

* Iar prin LSP se asigură o măsura a calităţii moştenirii, verificându-se ca prin moştenire să se obţină subclase care au aceleaşi proprietăţi ca şi superclasa.

Aceste trei principii sunt strâns legate între ele: dacă este încălcat unul dintre principiile LSP sau DIP, atunci implicit este încălcat OCP.

5.4. Stabilitate. Principiul dependenţelor stabile

Aşa cum s-a arătat în cadrul subcapitolului *1.2. Probleme ale software-ului*, **interdependenţa** reprezintă una dintre cauzele pentru care un design este rigid, imobil şi dificil de reutilizat.

Totuşi, interdependenţa este necesară dacă modulele implicate în design "colaborează".

Ca urmare există tipuri de **dependenţă utile** şi tipuri de **dependenţă indezirabile**.

În acest paragraf se propune un model de design în care dependenţele sunt toate utile şi se descrie un set de indicatori pentru măsurarea conformităţii designului cu modelul propus.

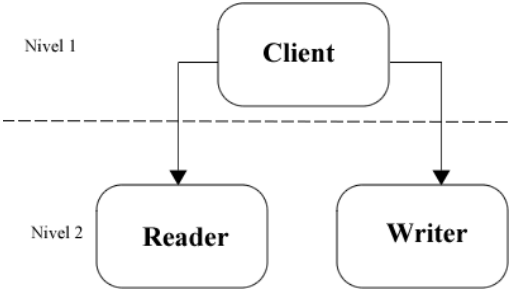
Aceşti indicatori măsoară stabilitatea software-ului.

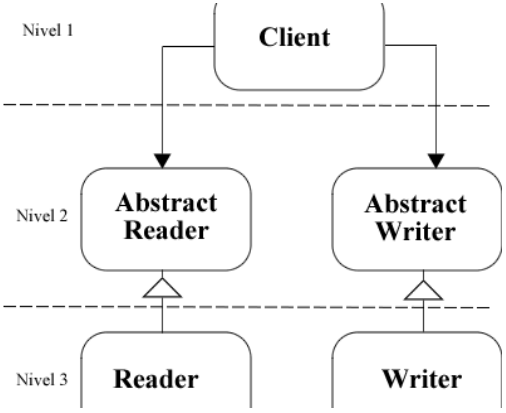
Stabilitatea este însăşi esenţa designului software.

Stabilitate şi Dependenţă. Dependenţe utile

Se consideră exemplul din *subcapitolul 5.3 (Principiul inversării dependenţelor),* cel al unui sistem alcătuit din 3 module: *Client*, *Reader*, *Writer* (Fig. 5.4).

Din analiza detaliată a acestei structuri, a rezultat că acest design este rigid, fragil şi dificil de reutilizat, din cauza faptului că modulul care implementează logica programului (modulul *Client*), este dependent de modulele de pe nivelul inferior. Pentru înlăturarea acestei dependenţe, se aplică DIP şi se obţine o structură (Fig. 5.5) în care modulul de pe nivelul superior depinde de nişte abstractizări. Designul astfel obţinut, este **robust, mentenabil şi uşor de reutilizat**.





Dependenţe utile

Totuşi, nu toate dependenţele au fost îndepărtate, ci doar cele care afectează calităţile programului.

-Dependenţele rămase sunt **nevolatile**, pentru că obiectul/modulul/clasa faţă de care se manifestă dependenţa este puţin probabil să se modifice.

- Probabilitatea ca aceste clase abstracte, AbstractReader şi AbstractWriter, să se modifice este foarte mică, spunem despre ele că au o **volatilitate scăzută**.

-Deoarece clasa Client depinde de module nevolatile, este puţin predispusă la schimbări.

- Această situaţie ilustrează foarte bine principiul deschis – închis: clasa *Client* este **deschisă la extinderi**, deoarece putem crea noi versiuni de clase concrete derivate din AbstractReader şi AbstractWriter pe care Client să le acţioneze, şi este **închisă pentru modificări** deoarece nu trebuie modificată pentru a face aceste extinderi.

Putem afirma în consecinţă, că o **dependenţă utilă** este o dependenţă faţă de un modul/clasă cu o volatilitate scăzută.

Cu cât volatilitatea este mai scăzută, cu atât dependenţa este "mai bună".

O dependenţă este indezirabilă când se manifestă faţă de un modul volatil.

Cu cât modulul/clasa faţă de care se manifestă dependenţa este mai volatil, cu atât dependenţa este "mai nedorită".

Stabilitatea

Volatilitatea unui modul depinde de mai multe tipuri de factori.

Spre exemplu, există programe care sunt publicate cu numărul de versiune; module care conţin numărul versiunii sunt volatile, deoarece sunt modificate ori de câte ori o nouă versiune este lansată.

Pe de altă parte, alte module sunt modificate mult mai rar.

Volatilitatea depinde şi de presiunea pieţei şi cererile clienţilor.

Un modul este sau nu posibil să fie modificat, dacă conţine sau nu ceva ce clientul doreşte să schimbe. Acest tip de factori sunt greu de apreciat.

Există, totuşi, un factor care influenţează volatilitatea şi care poate fi măsurat: **stabilitatea**. Stabilitatea, în sensul general acceptat, se defineşte ca fiind "**greu de modificat**".

Stabilitatea *nu este o măsura a probabilităţii de a modifica un modul, ci a dificultăţii de a-l modifica*.

Deci, modulele care sunt mai greu de modificat, sunt mai puţin volatile.

În exemplul amintit mai sus, clasele abstracte AbstractrReader şi AbstractWriter sunt stabile. Caracteristicile care fac aceste clase stabile, sunt:

- sunt **independente,** nu depind de ”nimic” şi ”nimic” nu poate forţa o schimbare a lor.

  *Se numesc clase **independente** acele clase care nu depind de nimic altceva.*

- de aceste clase depind alte clase (Client, Reader şi Writer depind de clasele abstracte din exemplu).

- Clasele derivate sunt clase dependente.
- Cu cât vor exista mai multe clase derivate, cu atât mai greu va fi să modificăm clasele de bază.
- Dacă vom dori să modificăm cele două clase abstracte, va trebui să modificăm şi clasele derivate.
- Deci, există motive serioase pentru care nu vom modifica cele două clase, îmbunătăţindu-le astfel stabilitatea.

*Clasele de care depind multe alte clase, se numesc **responsabile**.*

Clasele responsabile tind să fie stabile deoarece orice modificare a lor are un impact puternic asupra claselor dependente.

Cele mai stabile clase sunt cele care sunt independente şi responsabile, deoarece asemenea clase nu au nici un motiv să se modifice, şi au multe motive să nu se modifice.

Principiul dependenţelor stabile

Într-un proiect (design), dependenţa dintre pachete ar trebui să fie în sensul creşterii stabilităţii pachetelor. Un pachet ar trebui să depindă de pachete mai stabile decât el.

Un design nu poate fi complet static, un anume grad de volatilitate este necesar dacă se doreşte realizarea mentenanţei.

Acest lucru se obţine dacă designul se conformează *principiului închiderii generale* (Common Closure Principle, CCP).

Prin utilizarea acestui principiu, se creează pachete care sunt sensibile doar la anumite tipuri de modificări.

Aceste pachete *sunt proiectate să fie volatile*.

Modificările în aceste pachete sunt aşteptate şi prevăzute.

Nici un pachet, despre care ştim că va fi volatil, nu ar trebui să aibă între dependenţi pachete greu de modificat. În caz contrar, şi pachetul volatil va fi greu de modificat.

În conformitate cu principiul dependenţelor stabile pachetele care sunt proiectate să fie instabile (uşor de modificat) nu au ca dependenţi pachete care au o stabilitate mai mare decât a lor (mai greu de modificat).

Indicatori ai stabilităţii

O modalitate de a măsura stabilitatea unui pachet este numărarea dependenţelor care "intră" şi "ies" din pachet. Aceasta ne va ajuta la calcularea **stabilităţii poziţionale** a pachetului.

Se definesc următorii indicatori:

- *Ca* : **dependenţe aferente**: Numărul claselor din afara pachetului care depind de clasele din acest pachet;
- *Ce* : **dependenţe eferente**: Numărul de clase din cadrul pachetului care depind de clase din afara pachetului;
- *I* : **Instabilitatea**:

$$I = \frac{Ce}{Ca + Ce}$$

Acest indicator are valori în domeniul [0,1].

I = 0 indică un **grad maxim de stabilitate** a pachetului;

I = 1 indică un **grad maxim de instabilitate** a pachetului.

Indicatorii *Ca* şi *Ce* sunt calculaţi prin numărarea **claselor** din exteriorul pachetului în cauză care au relaţii de dependenţă cu clasele din interiorul pachetului.

Să considerăm următorul exemplul din **Fig. 5.11.,** în care săgeţile punctate reprezintă dependenţele între pachete.

Relaţiile dintre clasele pachetelor arată care este natura dependenţei, modul cum este implementată aceasta (moştenire, agregare, asociere).

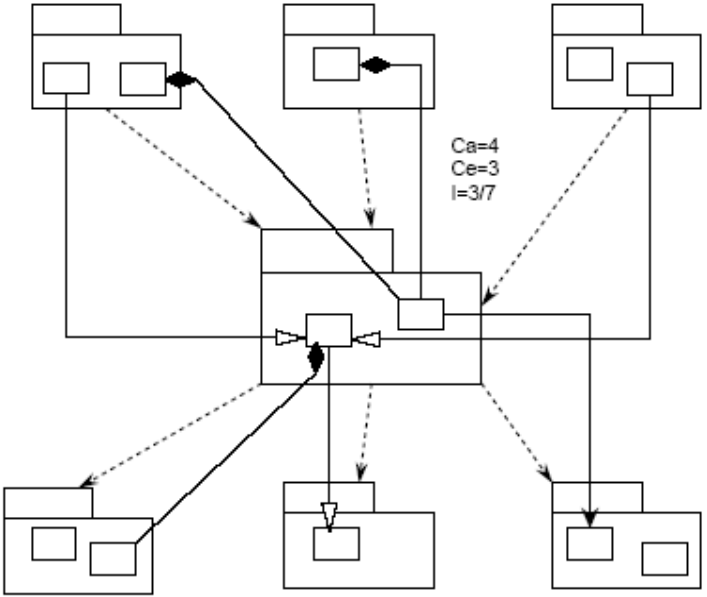


Fig. 5.11. Exemplu de calculare a gradului de stabilitate al unui pachet

Calcularea stabilităţii pachetului din centrul diagramei.

Observăm ca există 4 clase exterioare pachetului care sunt în relaţii cu clasele interioare pachetului, deci *Ca=4*. Mai mult, există 3 clase exterioare pachetului central de care depind clasele interioare, deci *Ce=3*.

$$I = \frac{Ce}{Ca + ce} = \frac{3}{7}$$

Când ***I=1***, nici un alt pachet nu depinde de pachetul curent, dar el depinde de alte pachete. Acesta este gradul maxim de instabilitate al unui pachet; pachetul este *iresponsabil* şi *dependent.*

Când ***I=0***, pachetul curent nu depinde de nici un alt pachet, dar de el depind alte pachete.

Este *responsabil* şi *independent*.

Un astfel de pachet are un grad maxim de stabilitate.

Din cauza pachetelor dependente, pachetul curent este dificil de modificat, şi nu depinde de alte pachete care ar putea forţa o modificare a acestuia.

Principiul dependenţelor stabile afirmă că indicatorul I al unui pachet ar trebui să fie mai mare decât indicatorul I al pachetelor dependente de el; I ar trebuie să descrească în sensul dependenţei.

Nu toate pachetele ar trebui să fie stabile. Dacă toate pachetele din sistem sunt stabile, atunci sistemul nu ar mai putea fi modificat. Sistemul trebuie proiectat astfel încât unele pachete să fie stabile iar altele instabile. **Fig. 5.12** arată situaţia ideala pentru un sistem de trei pachete.

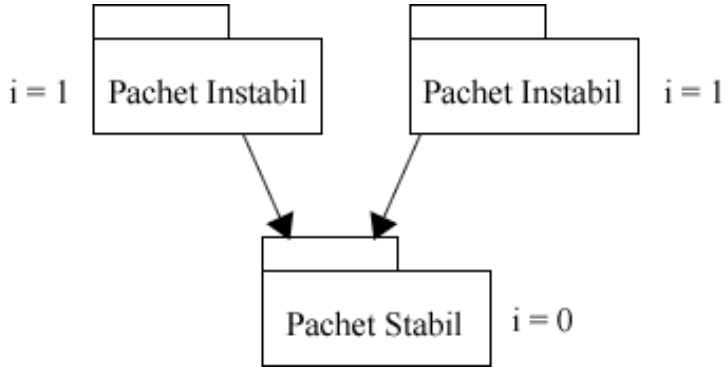


Fig. 5.12. Situaţia ideală din punct de vedere a stabilităţii pentru un sistem cu trei pachete

Pachetele care prezintă o probabilitate mai mare de a fi modificate sunt aşezate pe un nivel superior, iar cele care sunt stabile sunt aşezate la bază.

Pentru acest mod de aranjare a pachetelor, orice săgeată direcţionată în sus înseamnă o încălcare a principiului dependenţelor stabile.

O parte din software-ul unei aplicaţii nu trebuie să se modifice foarte des, şi anume logica de nivel înalt a aplicaţiei, deciziile de design.

Nu este de dorit ca aceste decizii arhitecturale să fie volatile, deci software-ul care încapsulează deciziile de nivel înalt, ar trebui plasat în cadrul unor pachete stabile.

Pachetele instabile ar trebui să conţină doar acele părţi de software, despre care se ştie că este probabil să fie modificate.

Dacă logica aplicaţiei este plasată în pachete stabile, atunci codul sursă al acestor pachete va fi dificil de modificat.

Acest fapt, ar putea face desing-ul inflexibil. OCP oferă soluţia pentru a face un pachet să fie flexibil şi totuşi să aibă un grad maxim de stabilitate (*I=0*),.

Conform acestui principiu este posibil şi este oportună crearea unor clase care să fie suficient de flexibile pentru a fi extinse fără modificări.

Acest lucru se realizează prin utilizarea claselor **abstracte**.

Principiul abstractizărilor stabile

Pachetele care au grad maxim de stabilitate ar trebui să fie maxim abstracte.

Pachetele instabile ar trebui să fie concrete.

Abstractizarea unui pachet ar trebui să fie direct proporţională cu stabilitatea sa.

Principiul abstractizărilor stabile (The Stable Abstractions Principle, SAP) stabileşte o **relaţie între stabilitate şi abstractizare** afirmând că un pachet stabil ar trebui să fie şi abstract astfel încât stabilitatea sa să nu-l împiedice să fie extins.

Pe de altă parte, un pachet instabil ar trebui să fie concret deoarece instabilitatea permite codului să fie cu uşurinţă modificat.

Dacă un pachet se doreşte a fi stabil, ar trebui să fie alcătuit din clase abstracte astfel încât să poată fi extins.

Pachetele stabile care pot fi extinse sunt şi flexibile şi nu constrâng design-ul.

Spre deosebire de **principiul inversări dependinţelor** care este un principiu pentru **clase**, **principiile dependenţelor stabile şi al abstractizărilor stabile** sunt principii care se aplică **pachetelor**.

Măsurarea gradului de abstractizare a unui pachet se face utilizând indicatorul *A,* care se calculează ca fiind raportul dintre numărul de clase abstracte dintr-un pachet şi numărul total de clase din pachet.

$$A = \frac{NumarClaseAbstracte}{NumarTotalClase}$$

Valorile indicatorului A sunt din intervalul [0,1].

Dacă ***A=0*** : înseamnă că un **pachet nu are clase abstracte**.

Dacă ***A=1*** : înseamnă ca **pachetul în cauză conţine numai clase abstracte**.

Relaţia dintre stabilitate, I, şi gradul de abstractizare, A (Fig. 5.13)

Pentru stabilirea relaţiei dintre stabilitate, măsurată de indicatorul *I*, şi gradul de abstractizare, măsurată cu indicatorul *A*, se realizează un grafic în care *A* se pune pe axa verticală, iar *I* pe axa orizontală.

Pachetele care au o stabilitate şi o abstractizare maximă se găsesc în punctul (0,1). Pachetele cu gradul de instabilitate cel mai mare şi concrete se găsesc în punctul (1,0).

Nu se poate impune tuturor pachetelor să se găsească fie la (0,1) fie la (1,0) deoarece pachetele au grade diferite de stabilitate şi abstractizare , aşa încât se admite că există un loc geometric al punctelor care definesc o poziţie rezonabilă pentru pachete în planul A/I.

Se deduce care este acest loc geometric, găsind ariile în care pachetele nu ar trebui să se găsească, adică ***zonele de excluziune***.

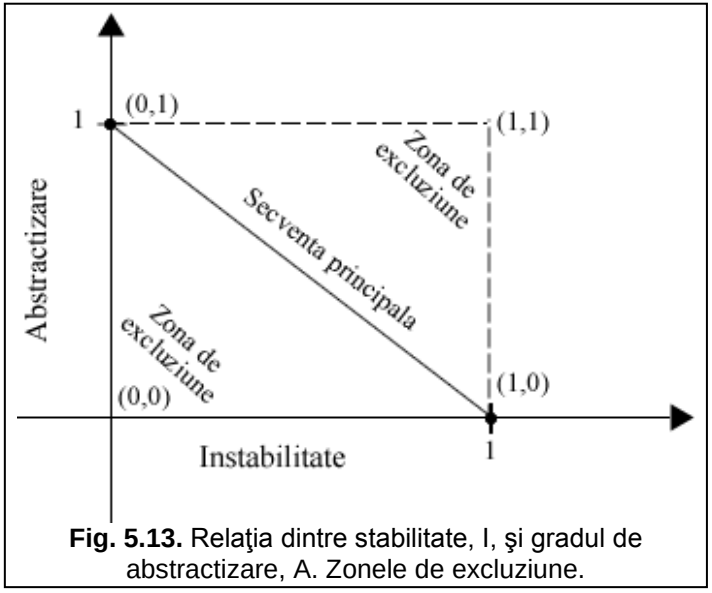


Fig. 5.13. Relaţia dintre stabilitate, I, şi gradul de abstractizare, A. Zonele de excluziune.

Se consideră un pachet în zona determinată de **A=0 şi I=0**, acesta va fi un **pachet stabil şi concret**.

Un astfel de pachet nu este de dorit deoarece este **rigid**, nu poate fi extins pentru că nu este abstract şi este foarte dificil de modificat pentru că este stabil.

Deci, nu este de dorit ca toate pachetele să se găsească în zona punctului (0,0). *Zona de vecinătate a punctului (0,0) este o zonă de excluziune.*

Se consideră un pachet în zona determinată de **A=1 şi I=1**.

Un astfel de tip de pachet este de asemenea indezirabil (poate chiar imposibil) deoarece are grad maxim de abstractizare şi totuşi nici un alt pachet nu depinde de el.

Şi acest tip de pachet este rigid, pentru că abstractizările sunt imposibil de extins.

Deci, un pachet in această zonă este fără sens.

Zona din jurul punctului (1,1) este o zonă de excluziune.

Fie pachetul **din A=0.5 şi I=0.5.** Acest pachet este parţial extensibil pentru că este parţial abstract; este parţial stabil deci extinderile nu duc la instabilitate maximă.

Un astfel de pachet pare **echilibrat**, deoarece *stabilitatea şi abstractizarea se compensează reciproc*.

Deci, **zona în care A=I nu este o zonă de excluziune.** Pe linia dintre (0,1) şi (1,0) se găsesc pachetele a căror abstractizare este compensată de către stabilitate.

Un pachet de pe această dreaptă nu este "prea abstract" pentru stabilitatea sa şi nici "prea instabil" pentru abstractizarea sa; are un număr "potrivit" de clase concrete şi abstracte, proporţional cu dependenţele aferente şi eferente.

Totuşi cel mai favorabil punct în care se poate găsi un pachet este la unul din cele două capete ale acestei drepte.

În practică, nu există un astfel de caz ideal.

Un pachet are caracteristici bune, dacă este plasat pe această dreaptă sau cât mai aproape de ea.

Distanţa faţă de Secvenţa Principală

Considerând că pe dreapta numită Secvenţa Principală, se găsesc pachetele care au caracteristicile ideale, putem crea un indicator care măsoară distanţa pachetului curent faţa de cazul ideal.

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

unde :

- D = Distanţa dintre punctul în care se găseşte pachetul faţă de dreapta numită secvenţa principală;
- A = gradul de abstractizare;
- I = gradul de stabilitate.

Acest indicator are valori cuprinse în intervalul [0, ~0.707]. (Putem normaliza acest indicator să aibă valori cuprinse în intervalul [0,1].)

Fiind dat acest indicator, un design poate fi analizat din punctul de vedere al conformării lui la **dreapta principală** . Pentru un pachet dat, D poate fi calculat. Orice pachet pentru care indicatorul D nu are o valoare apropiată de zero trebuie reexaminat şi restructurat.

Concluzii

Indicatorii descrişi mai sus măsoară conformitatea unui design faţă de un model de dependenţe şi abstractizări.

Aceşti indicatori încearcă să măsoare calitatea unui design din anumite puncte de vedere, fără a avea pretenţia de adevăr absolut.

În funcţie de dorinţe de la un design, se pot defini şi folosi diverşi indicatori.