Proiectarea Sistemelor Software Complexe

Curs 3 – Principii de Proiectare Orientată pe Obiecte

Principiile de proiectare orientată pe obiecte au fost formulate pentru a servi ca reguli pentru evitarea proiectării unei arhitecturi proaste. Principiile au fost propuse de către Robert Martin în lucrarea intitulată *Agile Software Development: Principles, Patterns, and Practices*[1]. În opinia lui Robert Martin există trei caracteristice care definesc o arhitectură proastă și care trebuie evitate:

- rigiditatea – sistemul este greu de modificat pentru că fiecare modificare afectează prea multe părți ale sistemului;
- fragilitatea – dacă se modifică ceva, apar tot felul de erori neașteptate;
- imobilitatea – este dificil să se reutilizeze părți dintr-o aplicație pentru că nu pot fi separate de aplicația pentru care au fost dezvoltate inițial.

1.1 Principiul Singurei Responsabilități (Single Responsibility Principle)

Enunț: *There should never be more than one reason for a class to change.*
(Niciodată nu trebuie să existe mai mult de un motiv pentru a modifica o clasă.)

În contextul acestui principiu prin responsabilitate se înțelege "un motiv de a modifica ". Dacă pot fi găsite mai multe motive pentru a modifica o clasă, înseamnă că acea clasă are mai multe responsabilități. Acest principiu spune faptul că o clasă nu trebuie să aibă mai multe responsabilități fiindcă orice modificare la nivelul cerințelor se reflectă printr-o modificare la nivelul uneia sau mai multor responsabilități care se propagă mai departe la nivelul claselor. Astfel, dacă o clasă implementează mai multe responsabilități automat pentru acea clasă la un moment dat va exista mai mult de un motiv pentru a fi modificată. În plus se poate ajunge la un efect de domino, și anume modificarea unei responsabilități poate duce la introducerea unor erori în implementarea corespunzătoare altei responsabilități. Se ajunge astfel la o interdependență nedorită între responsabilități.

În continuare principiul va fi ilustrat printr-un exemplu. În Fig. 3.1 este prezentată o diagramă de clase în care apare clasa *Rectangle* care are două metode: *draw()* și *area(): double*. Prima metodă este responsabilă de desenarea formei geometrice dreptunghi pe ecran, cea de a două calculează suprafața dreptunghiului.

În plus clasa *Rectangle* este folosită în două aplicații diferite. O aplicație face calcule geometrice: folosește clasa *Rectangle* doar pentru calcule matematice și nu folosește funcționalitatea de desenare. Cea de a două aplicație folosește atât metoda de desenare cât și cea ce calcul al suprafeței.

În acest exemplu clasa *Rectangle* încalcă principiul separării responsabilităților. Clasa are două responsabilități: (1) implementează modelul matematic al formei geometrice dreptunghi și (2) desenează dreptunghiul pe o interfață grafică.

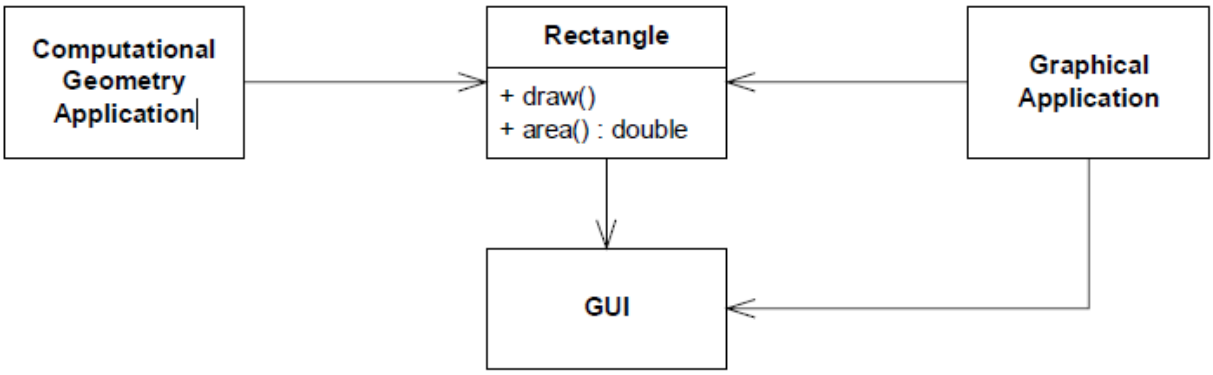

Fig. 3.1. Mai mult de o responsabilitate într-o singură clasă.

Violarea principiului separării responsabilităților în exemplul din Fig. 3.1 duce la apariția a două probleme. Având în vedre faptul că pentru a desena pe interfața grafică clasa *Rectangle* folosește clasa *GUI* este evident faptul că această clasă va trebui să fie disponibilă și în aplicația de calcule geometrice deși această aplicație nu folosește funcționalitatea de desenare. Acest lucru înseamnă timp de compilare mai lung și o dimensiunea mai are a executabilului. A doua problemă constă în faptul că, dacă clasa *Rectangle* este modificată pentru a rezolva o problemă în partea de desenare acest lucru va necesita nu doar recompilarea și retestarea aplicației de desenare ci și a celei de calcul geometric.

În Fig. 3.2 este prezentată diagrama de clase din Fig. 3.1 dar corectată astfel încât să nu existe mai mult de o responsabilitate pe clasă. Astfel a fost adăugată clasa *GeometricRectangle* care implementează modelul matematic al unui dreptunghi, obținându-se astfel o decuplare a celor două responsabilități. În acest nou design modificările făcute la partea de desenare nu mai influențează aplicația de calcule geometrice.

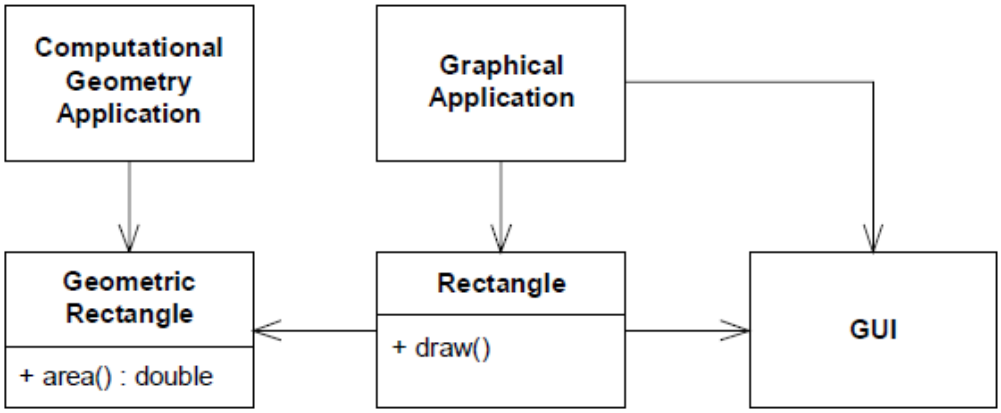

Fig. 3.2. O singură responsabilitate pentru fiecare clasă.

1.2 Principiul Deschis-Închis (Open-Close Principle)

Enunț: *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*
(Entitățile software (clasele, modulele, funcțiile etc.) trebuie să fie deschise în ceea ce privește extinderea, dar închise în ceea ce privește modificarea.)

Atunci când o singură modificare la un program generează o serie de modificări în toate module între care există o dependență, putem spune cu certitudine că acel program a fost proiectat incorect. Un astfel de program este fragil, rigid, impredictibil și nereutilizabil. *Principiul Închis-Deschis* are drept scop tocmai evitarea unei astfel de erori. El spune că trebuie proiectate module care nu se modifică niciodată. Atunci când cerințele se modifică trebuie extins comportamentul modulelor software prin adăugarea de cod nou și nu prin modificarea codului existent care a fost deja testat și este funcțional.

Modulele care respectă acest principiu au două proprietăți importante:

- **Sunt deschise pentru extindere**: adică, comportamentul modului poate fi extins; se poate actualiza modulul astfel încât să înglobeze noi comportamente pe măsură ce se modifică cerințele aplicației sau pentru a satisface nevoile unei alte aplicații.
- **Sunt închise pentru modificare**: adică, codul sursă al unui astfel de modul nu poate fi modificat.

Aparent cele două atribute par să se contrazică. Soluția la această problemă este abstractizarea. În cazul programării orientate pe obiect abstractizarea se poate obține prin intermediul claselor abstracte, iar diferitele comportamentele se obțin print derivarea claselor abstracte. Astfel un modul poate fi închis pentru modificări pentru că el depinde de o clasă abstractă care nu poate fi modificată, dar totuși comportamentul modului poate fi extins prin derivarea clasei abstracte.

În Fig. 3.3 este prezentată o diagramă de clase care nu respectă principiul deschis-închis. Atât clasa *Client* cât și clasa *Server* sunt clase concrete. Clasa *Client* folosește clasa *Server*. Dacă mai târziu se dorește ca, clasa Client să folosească un alt tip de server va fi nevoie să se modifice clasa *Client* astfel încât să utilizeze noul server.

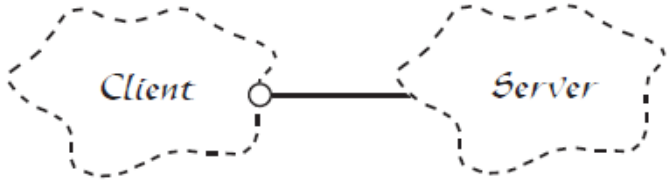

Fig. 3.3. Exemplu de clase care nu respectă principiul închis-deschis.

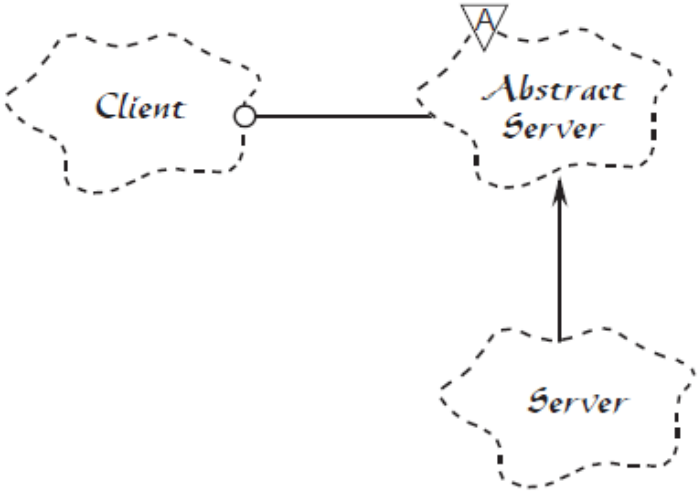

Fig. 3.4. Exemplu de clase care respectă principiul închis-deschis.

În Fig. 3.4 se prezintă același design ca și în Fig. 3.3, dar de această dată principiul deschis-închis este respectat. În acest caz a fost introdusă clasa abstractă *AbstractServer*, iar clasa *Client* folosește această abstractizare. Totuși în spate clasa Client va folosi de fapt clasa Server care implementează clasa *AbstractServer*. Dacă în viitor se dorește folosirea unui alt tip de server tot ce trebuie făcut va fi să se implementeze o nouă clasă derivată din clasa *AbstractServer*, de această dată clientul nu mai trebuie modificat.

1.3 Principiul Substituției Liskov (Liskov Substitution Principle)

Enunț: *Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.*
(Funcțiile care utilizează pointări sau referințe la clase de bază trebuie să poată folosi instanțe ale claselor derivate fără să își dea seama de acest lucru.)

Pentru a evidenția importanța acestui principiu se va considera o funcție care nu respectă acest principiu. Acest lucru înseamnă că funcția folosește o referință la o clasă de bază, dar că trebuie să știe care sunt toate clasele derivate din acea clasă de bază. Această metoda în mod evident încalcă principiul deschis-închis întrucât funcția trebuie modificată de fiecare data când este creată o nouă clasă derivată din clasa de bază.

În Fig. 3.5 este prezentată o diagramă de clase care nu respectă principiul substituției Liskov. Problema în această diagramă constă în faptul că, clasa *PersistentSet* deși este derivată din clasa abstractă Set nu poate folosi decât instanțe de clase derivate din clasa *PersientObject* pentru că doar acele obiecte pot fi scrise/citite într-un/dintr-un flux, în timp ce clasa *Set* poate folosi orice tip de obiect. De fiecare dată când o funcție adaugă un obiect intr-o instanța a clasei *Set* trebuie să verifice dacă clasa este de tipul *PersistentSet* pentru a filtra elementele adăugate în listă.

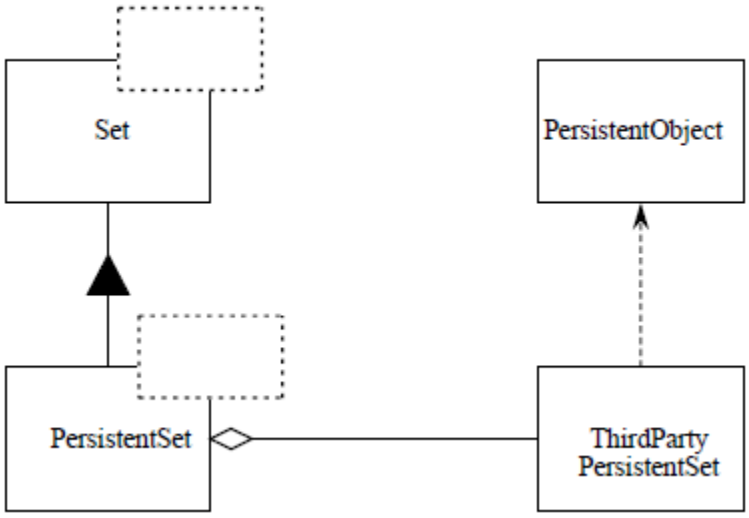

Fig 3.5. Diagramă de clase care încalcă principiul lui Liskov.

În Fig 3.6 este prezentată soluția pentru această problemă. Practic funcționalitatea comună claselor *Set* și *PersistentSet* a fost a fost pusă în două noi clase abstracte: *IterableContainer* și *MemberContainer*. De asemenea clasa *PersistentSet* numai este derivată din clasa Set; ambele clase având aceeași clasă de bază. Practic diferența între clasele *Set* și *PersistentSet* constă în ceea ce privește metoda care adaugă un

element, și anume, în cazul clasei Set este acceptat orice tip de obiect, în timp ce metoda de adăugare pentru clasa *PersistentSet* nu acceptă decât instanțe de clase derivate din clasa *PersistentObject*.

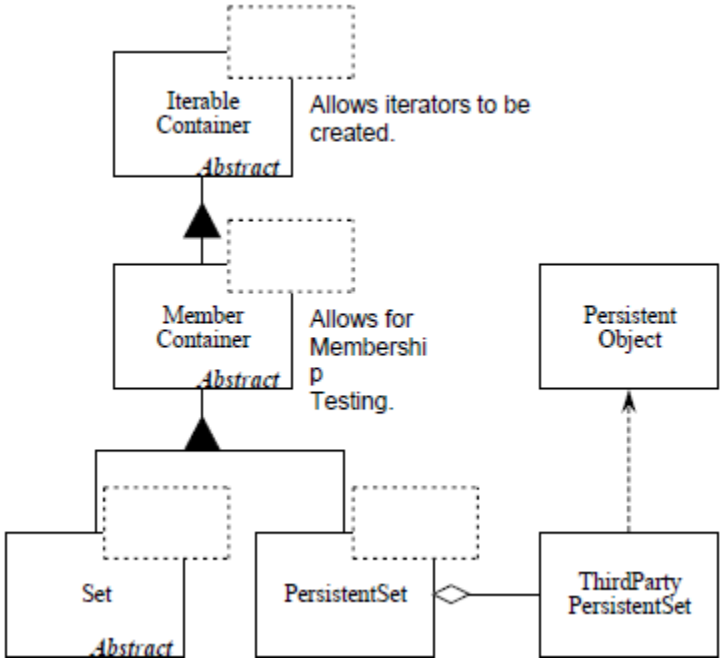

Fig 3.6. Diagramă de clase care nu încalcă principiul lui Liskov.

1.4 Principiul Segregării Interfeței (Interface Segregation Principle)

Enunț: *Clients should not be forced to depend upon interfaces that they don't use.*
(Clienții nu trebuie să depindă de interfețe pe care nu le folosesc)

Acest principiu scoate în evidență faptul că atunci când se definește o interfață trebuie avut grijă ca doar acele metode care sunt specifice interfeței să fie puse în interfață. Dacă într-o interfață sunt adăugate metode care nu am ce căuta acolo, atunci clasele care implementează interfața vor trebui să implementeze și acele metode. De exemplu, dacă se consideră interfața Muncitor care are metoda IaPrânzul, atunci toate clasele care implementează această interfața vor trebui să implementeze metoda IaPrânzul. Ce se întâmplă însă dacă muncitorul este un robot? Interfețele care conțin metode nespecifice se numesc interfețe poluate sau grase.

În Fig. 3.7 este prezentată o diagramă de clase care conține: interfața *TimerClient*, interfața *Door* și clasa *TimedDoor*. Interfața *TimerClient* trebuie implementată de orice clasă care are nevoie să intercepteze evenimente generate de un *Timer*. Interfața *Door* trebuie să fie implementată de orice clasă care implementează o ușă. Având în vedere că a fost nevoie de modelarea unei uși care se închide automat după un anumit interval de timp în Fig. 3.7 este prezentată o soluție în care a fost introdusă clasa *TimedDoor* derivată din interfața *Door*, iar pentru a dispune si de funcționalitatea din *TimerClient* a fost modificată interfața *Door* astfel încât să moștenească interfața *TimerClient*. Această soluție însă poluează interfața *Door* astfel că toate clasele care vor moșteni această interfața vor trebui să implementeze și funcționalitatea din *TimerClient*.

```
interface Door
{
        void Lock();
        void UnLock();
        IsDoorOpen();
}

class Timer
{
        public void Regsiter(int timeout, TimerClient client);
}

interface TimerClient
{
        void TimeOut();
};
```

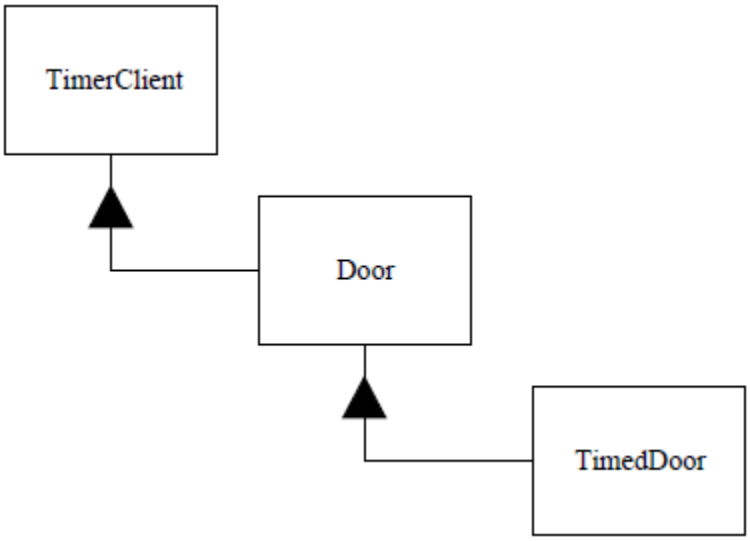

Fig. 3.7. Diagramă de clase care nu respectă principiul segregării interfețelor.

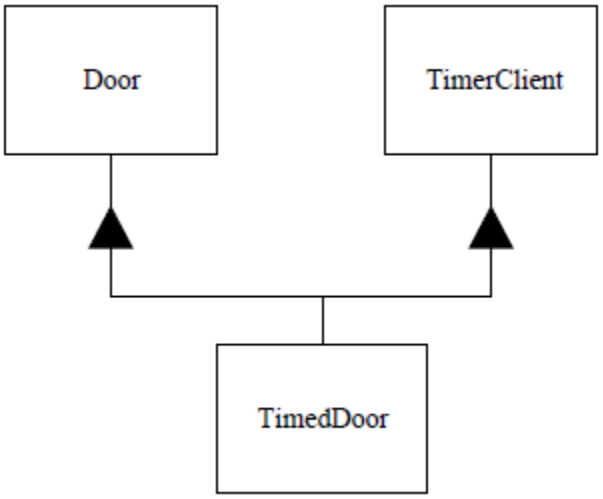

Fig. 3.8. Diagramă de clase care respectă principiul segregării interfețelor.

O soluție foarte simplă (Fig. 3.8) care respectă principiul segregării interfețelor constă în a deriva clasa *TimedDoor* atât din interfața *Door* cât și din interfața *TimerClient*. În acest fel clasa *TimedDoor* va implementa funcționalitatea dorită în timp ce celelalte clase care extind interfața *Door* nu vor trebui să implementeze funcționalități care nu sunt necesare.

1.5 Principiul Inversării Dependenței (Dependency Inversion Principle)

Enunț:

A) *High level modules should not depend upon low level modules. Both should depend upon abstractions.*
(Modulele de pe nivelurile ierarhice superioare nu trebuie să depindă de modulele de pe nivelurile ierarhice inferioare. Toate ar trebui să depindă doar de module abstract.)

B) *Abstractions should not depend upon details. Details should depend upon abstraction.*
(Abstractizările nu trebuie să depindă de detalii. Detaliile trebuie să depindă de abstractizări.)

Acest principiu spune faptul că modulele de pe nivelul ierarhic superior trebuie să fie decuplate de cele de pe nivelurile ierarhice inferioare, această decuplare realizându-se prin introducerea unui nivel de abstractizare între clasele care formează nivelul ierarhic superior și cele care formează nivelurile ierarhice inferioare. În plus principiul spune și faptul că abstractizarea nu trebuie să depindă de detalii, ci detaliile trebuie sa depindă de abstractizare. Acest principiu este foarte important pentru reutilizarea componentelor software. De asemenea, aplicarea corectă a acestui principiu face ca întreținerea codului să fie mult mai ușor de realizat.

În Fig. 3.9 este prezentată o diagramă de clase organizată pe trei niveluri. Astfel clasa *PolicyLayer* reprezintă nivelul ierarhic superior, ea accesează funcționalitate din clasa *MechanismLayer* aflată pe un nivel ierarhic inferior. La rândul ei clasa *MechanismLayer* accesează funcționalitate din clasa *UtilityLayer* care de asemenea se află pe un nivel ierarhic inferior. În concluzie, este evident faptul că în diagrama de clase prezentată nivelurile superioare depind de nivelurile inferioare. Acest lucru înseamnă că dacă apare o modificare la unul din nivelurile inferioare există șanse destul de mari ca modificarea să se propage în sus spre nivelurile ierarhice superioare. Ceea ce înseamnă că nivelurile superioare mai abstracte depind de nivelurile inferioare care sunt mai concrete. Așa dar se încalcă principiul inversări dependenței.

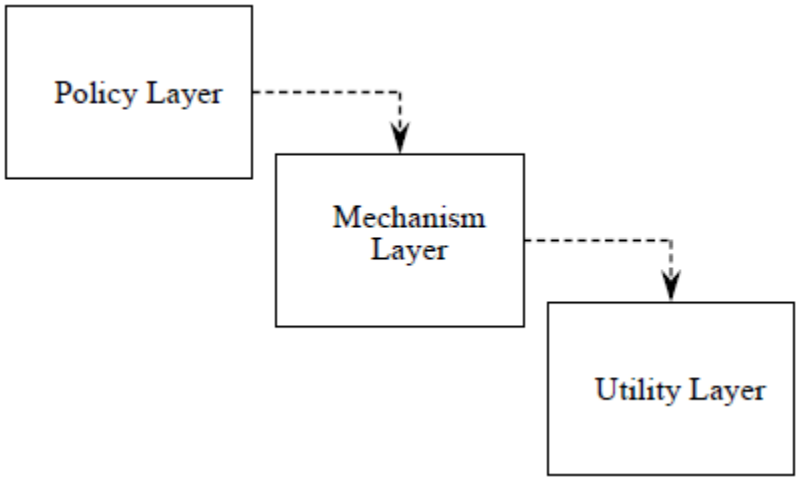

Fig. 3.9. Ierarhie de clase care nu respectă principiul inversării dependenței.

În Fig. 3.10 este prezentată aceeași diagramă de clase ca și în Fig. 3.9, dar de această dată este respectat principiul inversării dependenței. Astfel, la fiecare nivel care accesează funcționalitate dintr-un nivel ierarhic inferior a fost adăugată o interfață care va fi implementată de nivelul ierarhic inferior. În acest fel interfața prin care două niveluri comunică este definită în nivelul ierarhic superior astfel că dependența a fost inversată, și anume nivelul ierarhic inferior depinde de nivelul ierarhic superior. Modificări făcute la nivelurile inferioare nu mai afectează nivelurile superioare, ci invers. În concluzie diagrama de clase din Fig. 3.10 respectă principiul inversării dependenței.

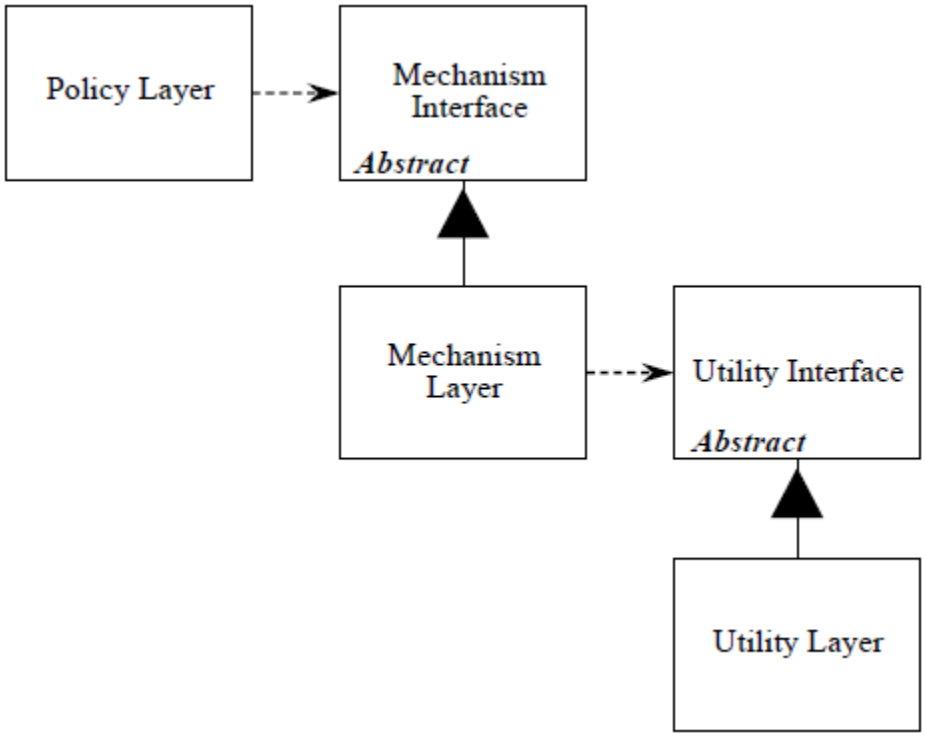

Fig. 3.10. Exemplu de clasă care respectă principiul inversării dependenței

Bibliografie

[1] Robert Martin. Agile Software Development: Principles, Patterns, and Practices. Editura Prentice Hall. 2002
[2] http://www.objectmentor.com/resources/articles/ocp.pdf
[3] http://www.objectmentor.com/resources/articles/lsp.pdf
[4] http://www.objectmentor.com/resources/articles/dip.pdf
[5] http://www.objectmentor.com/resources/articles/isp.pdf
[6] http://www.oodesign.com/design-principles.html