Inversion of control Container – Tema 2 PSSC | Mihaela ȚUȚU | semigrupa 5.2

Introducere:

Dependency Inversion Principle

Cand proiectam aplicatii software, consideram:

- **clase de nivel inferior** (*low level classes*) clasele care implementeaza operatii primare si de baza, precum accesul la disk, protocoale de retea;
- **clase de nivel superior** (*high level classes*) acele clase care implementeaza logica mai complexa.

Clasele de nivel superior se bazeaza pe clasele de nivel inferior. O cale de implementare a acestor structuri presupune implementarea claselor de nivel inferior si odata ce ele sunt implementate, sa se scrie clasele de nivel superior. Problema care apare in aceasta abordare este faptul ca acest *design* nu este flexibil. Ce s-ar intampla daca ar fi nevoie sa inlocuim clasele de nivel inferior?

Sa privim un exemplu classic cu un modul de copiere care citeste caractere de la tastatura si le scrie la un dispozitiv de imprimare. Clasa de nivel superior care contine logica este clasa *Copy.* Clasele de nivel inferior sunt clasele *KeyboardReader si PrinterWriter*.

Intr-un design gresit, clasa de nivel superior foloseste **direct** si **depinde intr-o mare masura de clasele de nivel inferior**. Intr-un astfel de caz, daca vrem sa redirectam iesirea catre o clasa *FileWriter*, trebuie sa facem schimbari in clasa *Copy*. (Presupunem ca aceasta clasa este foarte complexa, cu multe linii de cod si foarte dificil de testat).

Pentru a evita o astfel de situatie, putem introduce un alt nivel de abstractizare intre clasele de nivel superior si clasele de nivel inferior. Conform acestui principiu, vom avea:

Clase de nivel superior -> Un nou nivel de abstractizare -> Clase de nivel inferior *(High Level Classes --> Abstraction Layer --> Low Level Classes)*

Introducem principiul *Dependency Inversion*

- Modulele de nivel superior nu ar trebui sa depinda de modulele de nivel inferior. Amble tipuri de module trebuie sa depinde de abstractizari.
- Abstractizarile nu ar trebui sa depinda de detalii. Detaliile trebuie sa depinda de abstractizari.

Exemplu care incalca principiul *Dependecy Inversion:*

Avem o clasa Manager, care este o clasa de nivel superior si o clasa Worker, care este o clasa de nivel inferior. Adaugam o noua clasa aplicatiei, care va modela schimbarile in structura

companiei determinate de angajarea unor muncitori cu specializari noi. Pentru asta cream clasa SuperWorker. Presupunem clasa Manager o clasa cu o logica foarte complexa, pe care trebuie sa o modificam asa incat sa putem introduce noua clasa SuperWorker.

Dezavantajele acestei abordari:

- trebuie facute modificari in clasa Manager – o clasa complexa;
- cateva din actualele functionalitati ale clasei Manager ar putea fi afectate;
- testele unitare ar trebui refacute.

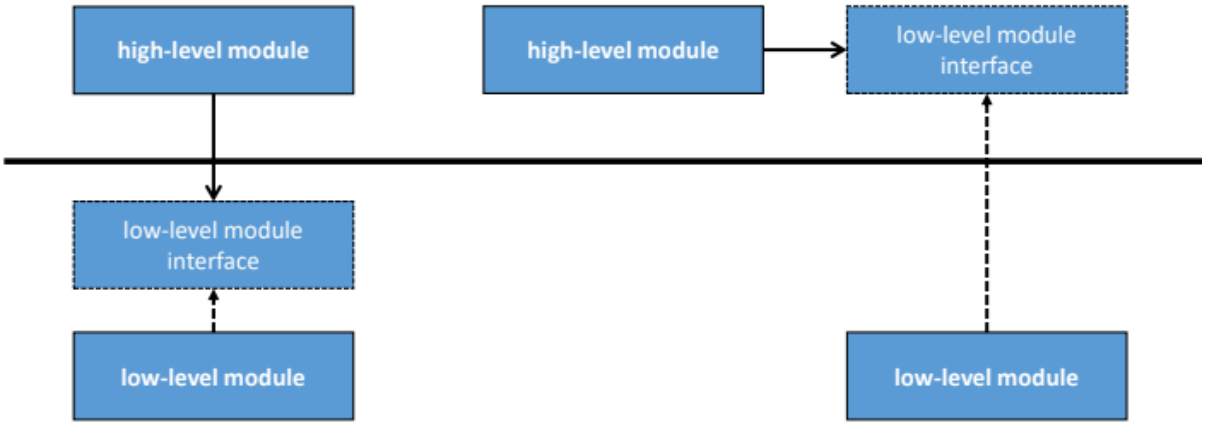
Aceste probleme sunt costisitoare ca timp si ar putea introduce noi erori in vechea functionalitate a aplicatiei. Lucrurile ar fi mai simple daca aplicatia ar fi structurata dupa principiul *Dependency Injection*, adica am proiecta *clasa Manager*, *o interfata IWorker (noul nivel de abstractizare)* si *o clasa Worker* care sa implementeze *interfata IWorker*. Cand ar trebui introdusa *clasa SuperWorker*, tot ce trebuie facut este sa implementam *interfata IWorker* corespunzatoare, fara alte modificari in clasele existente.

```
// Dependency Inversion Principle - Bad example

class Worker {

        public void work() {

                // ....working

        }

}



class Manager {

        Worker worker;



        public void setWorker(Worker w) {
                worker = w;
        }

        public void manage() {
                worker.work();
        }
}

class SuperWorker {
        public void work() {
                //.... working much more
        }
}
```

```
// Dependency Inversion Principle - Good example
interface IWorker {
        public void work();
}

class Worker implements IWorker{
        public void work() {
                // ....working
        }
}

class SuperWorker  implements IWorker{
        public void work() {
                //.... working much more
        }
}

class Manager {
        IWorker worker;

        public void setWorker(IWorker w) {
                worker = w;
        }

        public void manage() {
                worker.work();
        }
}
```

Cand acest principiu este aplicat, clasele de nivel superior nu interactioneaza direct cu clasele de nivel inferior, ci folosesc un nou nivel de abstractizare – *interfetele*. In acest caz, instantierea claselor de nivel inferior in interiorul clasei de nivel superior, se face fara operatorul *new*. Se pot folosi in schimb *design pattern-uri creationale* precum: *Factory Method*, *Abstract Factory*, *Prototype.*

Folosirea principiului *Dependency inversion* presupune un numar de clase si interfete care trebuie mentinute, cu alte cuvinte mai mult cod, dar mai flexibil. Acest principiu nu ar trebui sa se aplice claselor care raman neschimbate.



Definitie

Un IoC Container este o librarie care permite implementarea principiului Dependency Injection. *Dependency injection* este un tip de *Dependency inversion. (Dependency injection* != *Dependency inversion). IoC Container* este extem de folositor in a gestiona dependintele, in special atunci cand numarul lor este mare.

"Inversion of control – the approach of outsourcing the construction and management of objects."

Ideea principala este ca la inceputul realizarii unei aplicatii, se pot defini legaturi intre abstractizari si implamentari (de exemplu intre interfete si tipurile concrete). Apoi, *IoC Container* se va ocupa de crearea obiectelor si realizarea dependintelor.

Deoarece legaturile sunt configurate inca de la inceputul realizarii aplicatiei, totul de face intr-un loc si se poate vedea clar cum sunt conectate implementarile, in asa fel incat o eventuala schimabre nu ar trebui facuta decat intr-un loc. Pentru orice instanta a unei clase pe care *IoC Container* o face pentru noi, va face de asemenea si dependintele utilizate de acea clasa.

https://www.codeproject.com/Articles/615139/%2fArticles%2f615139%2fAn-Absolute-Beginners-Tutorial-on-Dependency-Inver

https://www.oodesign.com/dependency-inversion-principle.html

https://www.danclarke.com/all-about-inversion-of-control-containers/