Principiile SOLID

SOLID este un acronim compus din cinci principii de proiectare a claselor reprezentate de cinci principii de inginerie software care ajuta la crearea de clase mai robuste, mai flexibile si mai usor de intretinut. Aceste principii au fost dezvoltate de Robert C. Martin in anii 1990 pentru a ajuta la gestionarea complexitatii sistemelor si pentru a facilita refactorizarea si modificarea codului in timp.

Principiile sunt aplicate in timpul proiectarii arhitecturii sistemelor software, pentru a asigura ca clasele sunt structurate intr-un mod coerent si usor de inteles. Implementarea acestor principii poate duce la codebase-uri mai curate, mai usor de extins si mai putin vulnerabile la erori.

1. **S**ingle-responsibility principle (SRP)

Principiul responsabilitatii unice specifica ca o clasa trebuie sa aiba o singura responsabilitate si sa fie responsabila pentru un singur aspect al functionalitatii sistemului.

Martin defineste o responsabilitate ca fiind un motiv de schimbare, si concluzioneaza ca o clasa sau un modul ar trebui sa aiba un singur motiv de a fi schimbat (de exemplu, rescriere).

Ca exemplu, sa luam in considerare o clasa care compileaza si tipareste un raport. O astfel de clasa poate fi schimbata din doua motive. In primul rand, continutul raportului ar putea sa se schimbe sau in al doilea rand, formatul raportului ar putea sa se schimbe. Principiul responsabilitatii unice spune ca aceste doua aspecte ale problemei sunt de fapt doua responsabilitati separate si ar trebui sa fie, prin urmare, in clase separate. Ar fi o proiectare proasta sa cuplam doua lucruri care se schimba pentru motive diferite, in momente diferite.

Motivul pentru care este important sa pastram o clasa axata pe o singura preocupare este acela ca face clasa mai robusta. Continuand cu exemplul de mai sus, daca exista o schimbare in procesul de compilare a raportului, exista un risc mai mare ca in codul de imprimare sa apara erori daca este parte din aceeasi clasa.

2. **O**pen–closed principle (OCP)

Conform principiului deschis-inchis, clasele ar trebui sa fie deschise pentru extindere, dar inchise pentru modificare.

De exemplu, o clasa care implementeaza functionalitatea de trimitere a e-mailurilor ar trebui sa fie proiectata astfel incat sa poata fi extinsa in viitor pentru a trimite e-mailuri catre mai multi destinatari sau pentru a adauga un alt tip de mesaj. In loc sa modificam clasa existenta, ar trebui sa cream o noua clasa care extinde clasa existenta si sa adaugam functionalitatea suplimentara in acea noua clasa.

Acest principiu este important pentru a mentine coerenta si stabilitatea codului in timp si pentru a evita erorile sau problemele de mentenanta care ar putea aparea daca modificam o clasa existenta. Prin extinderea unei clase existente, putem adauga noi functionalitati fara a afecta codul existent si putem usura reutilizarea si modularizarea codului, ceea ce poate accelera dezvoltarea sistemelor software.

3. **L**iskov substitution principle (LSP)

Principiul substituirii lui Liskov se bazeaza pe idea ca obiectele din clasele derivate trebuie sa poata fi folosite in locul obiectelor din clasele de baza fara a afecta corectitudinea programului.

Acest principiu este important pentru a asigura interoperabilitatea intre obiectele de acelasi tip sau familie de obiecte si pentru a preveni erorile la nivelul programului. In caz contrar, o subclasa care nu respecta LSP poate introduce comportamente neasteptate sau erori in codul care foloseste superclasa, ceea ce poate duce la consecinte grave. Principiul substituirii Liskov este important pentru a asigura o arhitectura software robusta si usor de intretinut. Respectarea acestui principiu permite dezvoltatorilor sa creeze o ierarhie coerenta de clase si sa le utilizeze in mod eficient, astfel incat sistemul software sa poata fi extins fara probleme si sa permita o mai mare flexibilitate si adaptabilitate.

Abstractiile nu pot exista izolat, ele fiind mereu in relatie cu alte obiecte si componentele software din sistemul in care sunt utilizate. De asemenea aceastea pot sa nu reflecte complet si precis natura obiectelor si comportamentul acestora intr-un anumit context.

Violarea principiului substituirii Liskov este echivalenta cu incalcarea principiului deschis-inchis, deoarece o subclasa care nu respecta LSP poate duce la

modificari in clasa sa parinte si poate necesita modificari ale codului care foloseste clasa parinte, ceea ce duce la o inchidere a acesteia pentru extensibilitate.

4. **I**nterface segregation principle (ISP)

Principiul segregarii interfetelor propune ca interfetele trebuie sa fie mici si specializate, astfel incat sa fie posibila implementarea lor intr-un mod independent si reutilizabil.

Principiul segregarii interfetelor este important pentru a preveni crearea interfetelor inutile si pentru a incuraja crearea interfetelor specializate care sunt mai usor de utilizat si de intretinut. Respectarea acestui principiu ajuta la reducerea cuplarii intre clase si la crearea unor interfete mai clare si mai usor de inteles, care sunt mai eficiente si mai usor de intretinut.

De exemplu o interfata denumita "Mobilier" care contine metode generale pentru toate tipurile de mobilier. Daca aceasta interfata este implementata de diferite clase de mobilier, cum ar fi scaune, mese si dulapuri, acest lucru poate duce la crearea unor metode inutile sau ineficiente pentru anumite tipuri de mobilier, ceea ce ar putea duce la o implementare defectuoasa a acestora. O interfata separata pentru scaune ar putea contine metode specifice pentru a ajusta inaltimea, inclinarea si rotatia scaunului, in timp ce o interfata separata pentru dulapuri ar putea contine metode specifice pentru a deschide usile si pentru a adauga sau a elimina rafturi. Aceste interfete specializate sunt mai usor de inteles si de utilizat si permit clasele sa implementeze doar metodele relevante, fara a fi nevoie sa implementeze metode inutile sau ineficiente.

5. **D**ependency inversion principle (DIP)

Principiul inversarii dependentelor afirma ca clasele trebuie sa depinda de abstractiuni, nu de implementari. Astfel, dependentele sunt inversate intre clasele de nivel superior si cele de nivel inferior.

O aplicatie care utilizeaza o baza de date pentru a stoca si a accesa datele, in loc sa fie construita intr-un mod in care codul aplicatiei este direct dependent de codul specific al bazei de date, principiul DIP sugereaza crearea unei interfete abstracte care defineste modul in care aplicatia va interactiona cu baza de date. Astfel aplicatia nu este legata direct de o anumita implementare a bazei de date si este mai usor de intretinut. In plus, o astfel de abordare incurajeaza modularitatea si decuplarea intre diferitele parti ale aplicatiei, ceea ce face mai usor pentru dezvoltatorii sa schimbe sau sa inlocuiasca diferitele componente fara a afecta intreaga aplicatie.

Principiul DIP este important in proiectarea software-ului deoarece incurajeaza dezvoltarea de cod modular, flexibil si usor de intretinut. Respectarea acestui principiu

poate ajuta la prevenirea dependentelor nedorite intre diferite parti ale aplicatiei si poate facilita schimbarea sau inlocuirea diferitelor componente fara a afecta intregul sistem.

In concluzie, principiile SOLID sunt esentiale in proiectarea software-ului de calitate. Aceste principii ofera un set de ghiduri clare si concise pentru a crea cod modular, flexibil si usor de intretinut. In industria software, principiile SOLID sunt utilizate de dezvoltatori pentru a construi aplicatii fiabile, scalabile si usor de extins. Acestea sunt principii cheie care au fost testate si validate in timp prin experienta. Prin urmare, este important ca dezvoltatorii de software sa fie constienti de aceste principii si sa le aplice in mod constant in munca lor pentru a crea aplicatii robuste si de incredere.