SOLID PRINCIPI

Temelji kvalitetnog objektno orijentiranog dizajna

Programsko inženjerstvo → Vježbe

Što je SOLID?

S — **Single Responsibility**

Klasa ima samo jednu odgovornost i jedan razlog za promjenu

O — **Open / Closed**

Otvoreno za proširenja, zatvoreno za modifikacije

L — **Liskov Substitution**

Podklase moraju biti zamjenjive za svoje bazne klase

I — **Interface Segregation**

Više manjih sučelja umjesto jednog velikog

D — **Dependency Inversion**

Ovisnost o apstrakcijama, ne o konkretnim implementacijama

S Single Responsibility

"Klasa treba imati samo jedan razlog za promjenu."

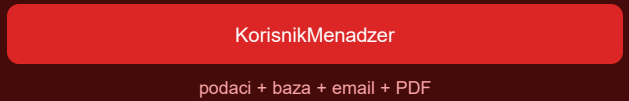
Definicija

Svaka klasa ili modul trebaju imati samo jednu odgovornost - jedan razlog za promjenu. Ako klasa ima više odgovornosti, promjena jedne može utjecati na druge.

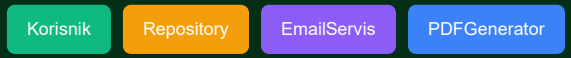
Prednosti

- Manje testnih slučajeva po klasi
- Manje ovisnosti između komponenti
- Lakše razumijevanje i održavanje koda
- Jednostavnije refaktoriranje

Loše: Više odgovornosti



Dobro: Jedna odgovornost



Ključno: Jedna promjena = jedna klasa

S Single Responsibility

LOŠE

```
// Klasa s previše odgovornosti!
public class Korisnik {
 private String ime, email;

// 1. Upravljanje podacima
 public void setIme(String ime) {
 this.ime = ime;
 }
 public String getIme() { return ime; }

// 2. Baza podataka
 public void spremiUBazu() {
 Connection conn = getConnection();
 ps.executeUpdate();
 }

// 3. Slanje emaila
 public void posaljiEmail() { email.send(); }

// 4. Generiranje izvještaja
 public String generirajPDF() { return "..."; }
 }
```

Problemi

- 4 razloga za promjenu klase
- Teško testiranje
- Visoka povezanost
- Kod se ne može ponovno koristiti

Što ako...

- Promijenimo bazu podataka?
- Trebamo drugi email servis?
- Format izvještaja se mijenja?

Jedna promjena = cijela klasa se mijenja

S Single Responsibility

DOBRO

```
// Samo podaci korisnika
public class Korisnik {
 private String ime;
 private String email;

 public String getIme() { return ime; }
 public void setIme(String ime) {
 this.ime = ime;
 }
 }
```

```
// Samo pristup bazi
public class KorisnikRepository {
 private Connection conn;

 public void spremi(Korisnik k) {
 PreparedStatement ps =
 conn.prepareStatement(...);
 ps.executeUpdate();
 }
 }
```

```
// Samo slanje emaila
public class EmailServis {

 public void posaljiDobrodoslicu(
 Korisnik korisnik) {
 Email email = new Email();
 email.setTo(korisnik.getEmail());
 email.send();
 }
 }
```

```
// Samo generiranje izvještaja
public class IzvjestajGenerator {

 public String generiraj(
 Korisnik korisnik) {
 return "PDF izvještaj za: "
 + korisnik.getIme();
 }
 }
```

O Open / Closed

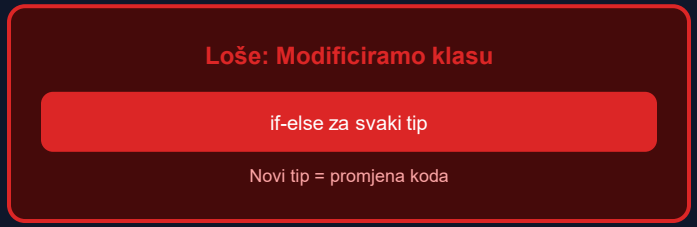
"Otvoreno za proširenje, zatvoreno za modifikacije."

Definicija

Softverski entiteti (klase, moduli, funkcije) trebaju biti otvoreni za proširenje, ali zatvoreni za modifikacije. Nova funkcionalnost se dodaje bez mijenjanja postojećeg koda.

Kako postići?

- Korištenje apstrakcija (interfaces, abstract classes)
- Polimorfizam umjesto if-else lanaca
- Strategy pattern, Factory pattern
- Dependency Injection



Ključno: Dodajemo klase, ne mijenjamo postojeće

O Open / Closed

LOŠE

```
// Dodavanje novog oblika zahtijeva promjenu koda!
public class KalkulatorPovrsine {

 public double izracunaj(Object oblik) {


if (oblik instanceof Pravokutnik) {
 Pravokutnik p = (Pravokutnik) oblik;
 return p.sirina * p.visina;
 }


else if (oblik instanceof Krug) {
 Krug k = (Krug) oblik;
 return Math.PI * k.radius * k.radius;
 }


// Novi oblik? Moramo dodati novi else if!

else if (oblik instanceof Trokut) {
 // ... nova logika
 }

 return 0;
 }
 }
```

Problemi

Svaki novi oblik = promjena klase

Rastući if-else lanac

Rizik od regresije

Teško održavanje

Kršenje OCP

Klasa NIJE zatvorena za modifikacije - svaki put kad dodamo novi oblik, moramo mijenjati postojeći kod.

instanceof + **if-else** = code smell

O Open / Closed

DOBRO

```
// Apstrakcija - sučelje
public interface Oblik {
 double izracunajPovrsinu();
 }
```

```
public class Pravokutnik implements Oblik {
 private double sirina, visina;
 public double izracunajPovrsinu() {
 return sirina * visina;
 }
 }
```

```
public class Krug implements Oblik {
 private double radius;
 public double izracunajPovrsinu() {
 return Math.PI * radius * radius;
 }
 }
```

```
// Novi oblik - bez promjene koda!
public class Trokut implements Oblik {
 private double baza, visina;
 public double izracunajPovrsinu() {
 return (baza * visina) / 2;
 }
 }
```

```
// Kalkulator - nikad se ne mijenja!
public class KalkulatorPovrsine {
 public double izracunaj(Oblik o) {
 return o.izracunajPovrsinu();
 }
 }
```

Arhitektura

Rezultat

Novi oblici bez promjena koda

Polimorfizam u akciji

L Liskov Substitution

"Podklase moraju biti zamjenjive za svoje bazne klase."

Definicija

Ako je klasa B podklasa klase A, tada objekte klase A možemo zamijeniti objektima klase B bez narušavanja ispravnosti programa.

Pravila

Podklasa ne smije suziti ponašanje bazne klase

Preduvjeti ne smiju biti jači u podklasi

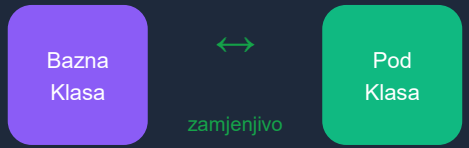
Postuvjeti ne smiju biti slabiji u podklasi

Invarijante moraju biti očuvane

Klasičan primjer kršenja

Kvadrat nasljeđuje Pravokutnik - ali kvadrat ima drugačija ograničenja (a=b), što može srušiti kod koji očekuje pravokutnik.

Princip zamjenjivosti



Ključno: Program mora raditi jednako ispravno bilo da koristi objekt bazne klase ili podklase

L Liskov Substitution

LOŠE

```
public class Pravokutnik {
 protected int sirina, visina;
 public void setSirina(int s) { this.sirina = s; }
 public void setVisina(int v) { this.visina = v; }
 public int getPovrsina() { return sirina * visina; }
 }
```

```
// Kvadrat "je" pravokutnik - ili nije?
public class Kvadrat extends Pravokutnik {
 @Override
 public void setSirina(int s) {
 this.sirina = s;
 this.visina = s;
// Neočekivano!
 }
 @Override
 public void setVisina(int v) {
 this.visina = v;
 this.sirina = v;
// Neočekivano!
 }
```

```
// Ovaj kod NE radi s Kvadratom!
void testirajPravokutnik(Pravokutnik p) {
 p.setSirina(5); p.setVisina(4);
 assert p.getPovrsina() == 20;
// FAIL!
 } // Kvadrat: 4x4=16, nikad 20!
```

Problem

Matematički: Kvadrat JE pravokutnik

Programski: NIJE zamjenjiv!

Očekivano vs. Stvarno

Pouka: Nasljeđivanje nije uvijek pravi izbor

L Liskov Substitution

DOBRO

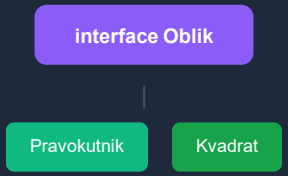
```
// Zajedničko sučelje
public interface Oblik {
 double getPovrsina();
 double getOpseg();
 }
```

```
public class Pravokutnik implements Oblik {
 private final int sirina, visina;
// immutable
 public Pravokutnik(int s, int v) { ... }
 public double getPovrsina() { return sirina*visina; }
 public double getOpseg() { return 2*(sirina+visina); }
 }
```

```
// Kvadrat - vlastita implementacija!
public class Kvadrat implements Oblik {
 private final int stranica;
 public Kvadrat(int s) { this.stranica = s; }
 public double getPovrsina() { return stranica*stranica; }
 public double getOpseg() { return 4*stranica; }
 }
```

```
// Ovo radi s BILO kojim oblikom!
void ispisiInfo(Oblik o) {
 System.out.println("Površina: " + o.getPovrsina());
 }
 ispisiInfo(new Pravokutnik(5, 4));
// OK!
 ispisiInfo(new Kvadrat(5));
// OK!
```

Ispravna hijerarhija



Zašto ovo radi

- Immutable objekti - nema settera
- Kompozicija umjesto nasljeđivanja
- Sučelje definira ponašanje

Potpuna zamjenjivost - svaki Oblik radi jednako

I Interface Segregation

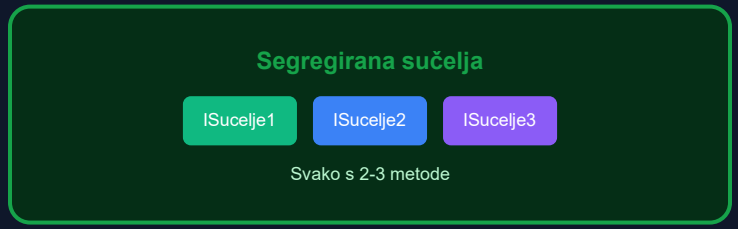
"Klijenti ne smiju ovisiti o metodama koje ne koriste."

Definicija

Bolje je imati više manjih, specifičnih sučelja nego jedno veliko opće sučelje. Klase ne bi trebale biti prisiljene implementirati metode koje ne koriste.

Prednosti

- Manje ovisnosti između komponenti
- Lakše testiranje i održavanje
- Fleksibilnije dizajn
- Nema praznih implementacija



I Interface Segregation

LOŠE

```
// "Fat interface" - previše metoda!
public interface IMultifunkcionalniUredaj {
 void ispisi(Dokument d);
 void skeniraj(Dokument d);
 void faksiraj(Dokument d);
 void kopiraj(Dokument d);
 void posaljiEmail(Dokument d);
 }
```

```
// Moderni printer - sve koristi
public class ModerniPrinter
 implements IMultifunkcionalniUredaj {
 public void ispisi(Dokument d) {
/* OK */ }
 public void skeniraj(Dokument d) {
/* OK */ }
 public void faksiraj(Dokument d) {
/* OK */ }
 // ... sve metode implementirane
 }
```

```
// Stari printer - MORA implementirati SVE!
public class StariPrinter
 implements IMultifunkcionalniUredaj {
 public void ispisi(Dokument d) {
/* OK */ }
 public void skeniraj(Dokument d) {

throw new UnsupportedOperationException();
 }
 // ... prazne metode ili iznimke
 }
```

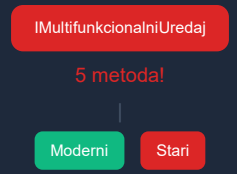
Problemi

Klase ovise o metodama koje ne koriste

Prazne implementacije ili iznimke

Kršenje LSP principa

Fat Interface



Stari printer MORA implementirati metode koje **nikad neće koristiti**

I Interface Segregation

DOBRO

```
public interface IPrinter {
 void ispisi(Dokument d);
 }
```

```
public interface IScanner {
 void skeniraj(Dokument d);
 }
```

```
public interface IFax {
 void faksiraj(Dokument d);
 }
```

```
// Implementira SVE što treba
public class ModerniPrinter
 implements IPrinter, IScanner, IFax {
 public void ispisi(Dokument d) { ... }
 public void skeniraj(Dokument d) { ... }
 public void faksiraj(Dokument d) { ... }
 }
```

```
// Implementira SAMO što može
public class StariPrinter implements IPrinter {
 public void ispisi(Dokument d) { ... }

// Nema praznih metoda!
 }
```

Prednosti

- Klase ovise samo o onome što koriste
- Nema praznih implementacija
- Lakše testiranje i održavanje

Više manjih sučelja umjesto jednog velikog

D Dependency Inversion

"Ovisi o apstrakcijama, ne o konkretnim implementacijama."

Definicija

High-level moduli ne smiju ovisiti o low-level modulima. Oba trebaju ovisiti o apstrakcijama. Apstrakcije ne smiju ovisiti o detaljima - detalji ovise o apstrakcijama.

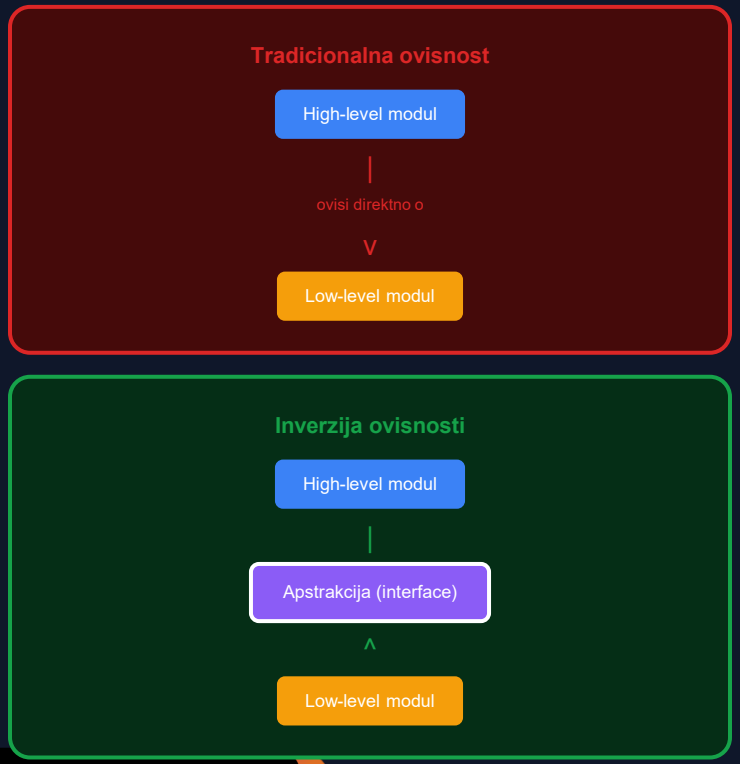
Prednosti

Labava povezanost (loose coupling)

Jednostavno testiranje s mock objektima

Laka zamjena implementacija

Dependency Injection pattern



D Dependency Inversion

LOŠE

```
// Konkretna implementacija
public class MySQLBaza {
 public void spremi(String podaci) {
 // MySQL specifična logika
 connection.execute("INSERT...");
 }
 }
```

```
// Direktna ovisnost o konkretnoj klasi!
public class KorisnikServis {

// Čvrsta veza s MySQLBaza!
 private MySQLBaza baza = new MySQLBaza();

 public void registriraj(Korisnik k) {
 // Validacija...
 baza.spremi(k.toString());
 }
 }
```

```
// Kako testirati bez prave baze?!
@Test
 void testRegistracije() {
 KorisnikServis servis = new KorisnikServis();

// Automatski kreira MySQLBaza!

// Ne možemo koristiti mock!
 servis.registriraj(korisnik);
 }
```

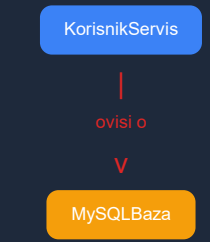
Problemi

Čvrsta veza s konkretnom klasom

Nemoguće unit testiranje

Promjena baze = promjena koda

Direktna ovisnost



High-level modul **ovisi** o low-level modulu

D Dependency Inversion

DOBRO

```
// Apstrakcija
public interface IBaza {
 void spremi(String podaci);
 String dohvati(int id);
 }
```

```
public class MySQLBaza implements IBaza {
 public void spremi(String p) {
/* MySQL */ }
 public String dohvati(int id) { ... }
 }
```

```
public class MongoDBBaza implements IBaza {
 public void spremi(String p) {
/* Mongo */ }
 public String dohvati(int id) { ... }
 }
```

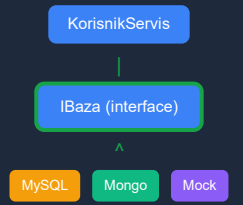
```
// Dependency Injection!
public class KorisnikServis {
 private final
IBaza baza; // Apstrakcija!

 public KorisnikServis(
IBaza baza) {
 this.baza = baza;
// Injektirano!
 }
 public void registriraj(Korisnik k) {
 baza.spremi(k.toString());
 }
 }
```

```
// Korištenje - lako zamijeniti!
// Produkcija
 IBaza baza = new MySQLBaza();
 KorisnikServis servis = new KorisnikServis(baza);


// Testiranje
 IBaza mockBaza = mock(IBaza.class);
 KorisnikServis test = new KorisnikServis(mockBaza);
```

Inverzija ovisnosti



Prednosti

- Labava povezanost
- Jednostavno testiranje s mock objektima
- Laka zamjena implementacija

Zašto primjenjivati SOLID?

Kvaliteta koda

Čišći i razumljiviji kod

Manje bugova

Bolja organizacija

Jasne odgovornosti

Održavanje

Lakše dodavanje značajki

Jednostavnije refaktoriranje

Manje rizika od regresije

Dugoročna održivost

Testiranje

Jednostavno unit testiranje

Mockanje ovisnosti

Izolacija komponenti

Veća pouzdanost

SOLID = Temelj profesionalnog razvoja softvera

Sažetak SOLID principa

S	O	L	I	D
Single Responsibility	**Open / Closed**	**Liskov Substitution**	**Interface Segregation**	**Dependency Inversion**
Jedna klasa = jedna odgovornost	Otvoreno za proširenje, zatvoreno za izmjene	Podklase zamjenjive za bazne klase	Više manjih sučelja umjesto jednog velikog	Ovisi o apstrakcijama, ne implementacijama
Razdvoji odgovornosti	Proširi, ne mijenjaj	Potpuna zamjenjivost	Podijeli sučelja	Injektiraj ovisnosti

Zajedno čine temelj kvalitetnog OOP dizajna!

Projektni zadatak

Zadatak

Implementirati sve SOLID principe u vlastiti projektni rad. Svaki član tima mora demonstrirati primjenu svih pet principa u svojem dijelu koda. Sve promjene potrebno je raditi na zasebnim „feature branchevima” i sve zajedno je potrebno „mergeati” u „master branch”. Obvezno deployati aplikaciju na neki dostupan server. Bodovi: I3 – 1 bod; I8 – 1 bod;

Što trebate napraviti:

1. Identificirati dijelove koda koji krše SOLID principe
2. Refaktorirati kod primjenom SOLID principa
3. Dokumentirati primjere i objasniti promjene
4. Prezentirati prednosti korištenja SOLID-a

Checklist za svaki princip

- **S** Razdvojene odgovornosti klasa
- **O** Proširive klase bez modifikacija
- **L** Ispravno nasljeđivanje
- **I** Segregirana sučelja
- **D** Dependency Injection

Cilj: Razumjeti i primijeniti SOLID u praksi