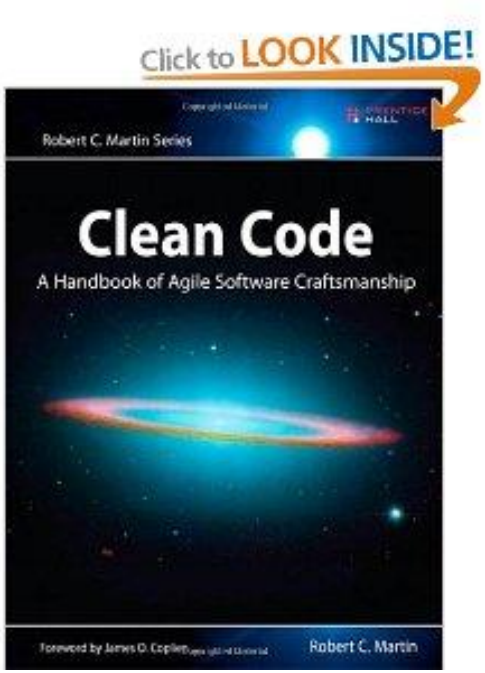
SOLID principi dizajna

Sveučilište u Zagrebu
Fakultet organizacije i informatike
Analiza i razvoj programa

Zlatko Stapić

SOLID

- Uveo ga je Robert C. Martins ("Uncle Bob")
  - Manifest agilnog razvoja
  - Autor nekolicine knjiga , npr. „Clean code”



SOLID

- SOLID
  - **S**ingle Responsibility Principle
  - **O**pen Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inverison Principle

Single Responsibility Principle

- *"Nikad ne bi trebalo postojati više od jednog razloga za promjenu klase." — Robert Martin, SRP dokument povezan s The Principles of OOD*
- Klasa bi trebala raditi jednu i samo jednu stvar

> "One class should have one and only one responsibility"

Single Responsibility Principle

- Jednostavan primjer:

Person.java

```
public class Person
{
    private Long personId;
    private String firstName;
    private String lastName;
    private String age;
    private List<Account> accounts;
}
```

Account.java

```
public class Account
{
    private Long guid;
    private String accountNumber;
    private String accountName;
    private String status;
    private String type;
}
```

Izvor: https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/

Single Responsibility Principle

- Još jedan primjer:

```
class Product {
  constructor(title, price, taxRate) {
    this.title = title;
    this.price = price;
    this.taxRate = taxRate;
  }
  calculateTax() {
    return this.price * this.taxRate;
  }
}

const table = new Product('a nice table', 55, 0.1);
console.log(table.calculateTax(table));
// output: 5.5
```

- Je li ova klasa u skladu sa SRP-om?

Izvor: https://muatik.medium.com/oop-solid-with-examples-d3dc310d72c3

Single Responsibility Principle

- Što bi bilo rješenje?
- Porezni kalkulator trebao bi biti izdvojen u zasebnu klasu...

```
class Product {
  constructor(title, price, taxRate) {
    this.title = title;
    this.price = price;
    this.taxRate = taxRate;
  }

getPrice() {
    return this.price;
  }

  getTaxRate() {
    return this.taxRate;
  }
}

class TaxCalculator {
  static calculateTax(product) {
    return product.getPrice() * product.getTaxRate();
  }
}

const table = new Product('a nice table', 55, 0.1);
console.log(TaxCalculator.calculateTax(table))
```

- Je li ovo u redu?

Izvor: https://muatik.medium.com/oop-solid-with-examples-d3dc310d72c3

Single Responsibility Principle

- Što bi bilo rješenje ?
- Poreznu stope bi trebale također biti izvađene jer su različite po zemljama…

```
class Product {
  constructor(title, price) {
    this.title = title;
    this.price = price;
  }

getPrice() {
    return this.price;
  }
}

const TAX_RATES = {
  "DE": {
    rate: 0.10,
    limit: 0
  },
  "UK": {
    rate: 0.12,
    limit: 60
  },
  "US": {
    rate: 0.11,
    limit: 0
  }
}

class TaxCalculator {
  static calculateTax(product, taxConfig) {
    return taxConfig.limit < product.getPrice() ? product.getPrice() *
taxConfig.rate : 0
  }
}
```

Izvor: https://muatik.medium.com/oop-solid-with-examples-d3dc310d72c3

Single Responsibility Principle

- Dvije odgovornosti

```
interface Modem {
 public void dial(String pno);
 public void hangup();

 public void send(char c);
 public char recv();
}
```

- Upravljanje vezom + komunikacija podacima

Izvor: Enbohm A. Solid – OO Design Principles

Single Responsibility Principle

- Odvojite u dva sučelja

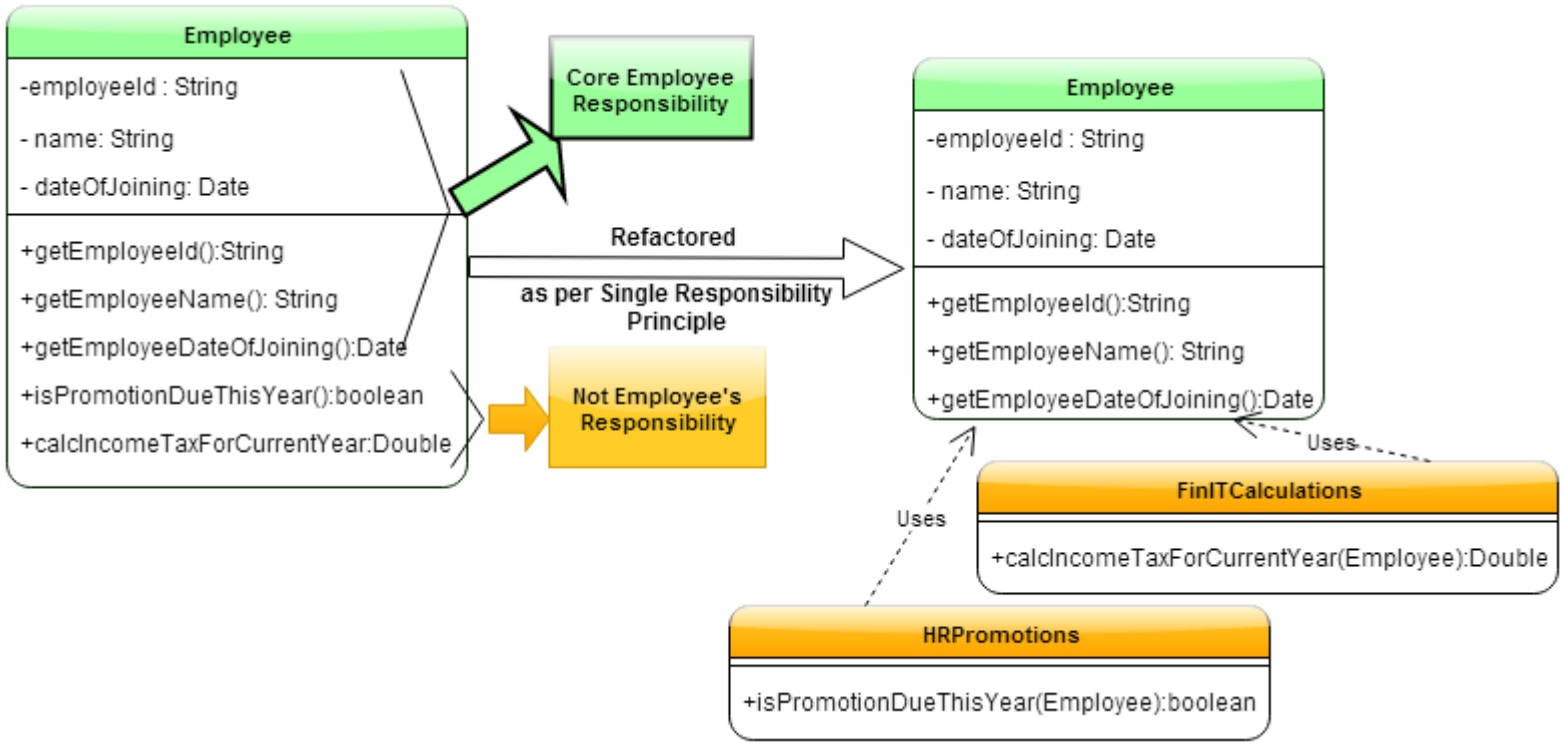
```
interface DataChannel {
 public void send(char c);
 public char recv();
}
```

```
interface Connection {
 public void dial(String phn);
 public char hangup();
}
```

Izvor: Enbohm A. Solid – OO Design Principles

Single Responsibility Principle

- Još jedan primjer:



Izvor: https://www.javabrahman.com/programming-principles/single-responsibility-principle-with-example-in-java/

Open Closed Principle

- *"Softverski entiteti (klase, moduli, funkcije itd.) trebaju biti otvoreni za proširenje, ali zatvoreni za izmjene." — Robert Martin parafrazirajući Bertranda Meyera, OCP dokument s poveznicom iz The Principles of OOD*

- Promijenite ponašanje klase korištenjem nasljeđivanja i kompozicije:

> "Software components should be open for extension, but closed for modification"

Open Closed Principle

```
// Open-Close Principle – Loš primjer
 class GraphicEditor {

 public void drawShape(Shape s) {
          if (s.m_type==1)
                    drawRectangle(s);
          else if (s.m_type==2)
                    drawCircle(s);
          }
          public void drawCircle(Circle r) {....}
          public void drawRectangle(Rectangle r) {....}
 }

 class Shape {
          int m_type;
 }

 class Rectangle extends Shape {
          Rectangle() {
                    super.m_type=1;
          }
 }

 class Circle extends Shape {
          Circle() {
                    super.m_type=2;
          }
 }
```

Izvor: https://www.oodesign.com/design-principles.html

Open Closed Principle

Problemi

- Nije moguće dodati novi `oblik` bez izmjene `GraphEditora`
- Važno je razumjeti `GraphEditor` za dodavanje novog `oblika`
- Čvrsta veza između `GraphEditora` i `Shapea`
- Teško je testirati određeni `oblik` bez uključivanja `GraphEditora`
- `If-Else-/Case` treba izbjegavati

Izvor: Enbohm A. Solid – OO Design Principles

Open Closed Principle - poboljšano

```
// Princip Open-Close - Dobar primjer
class GraphicEditor {
    public void drawShape(Shape s) {
        s.draw();
    }
}

class Shape {
    abstract void draw();
}

class Rectangle extends Shape {
    public void draw() {
    // draw the rectangle
    }
}
```

Izvor: https://www.oodesign.com/design-principles.html

Open Closed Principle - poboljšano

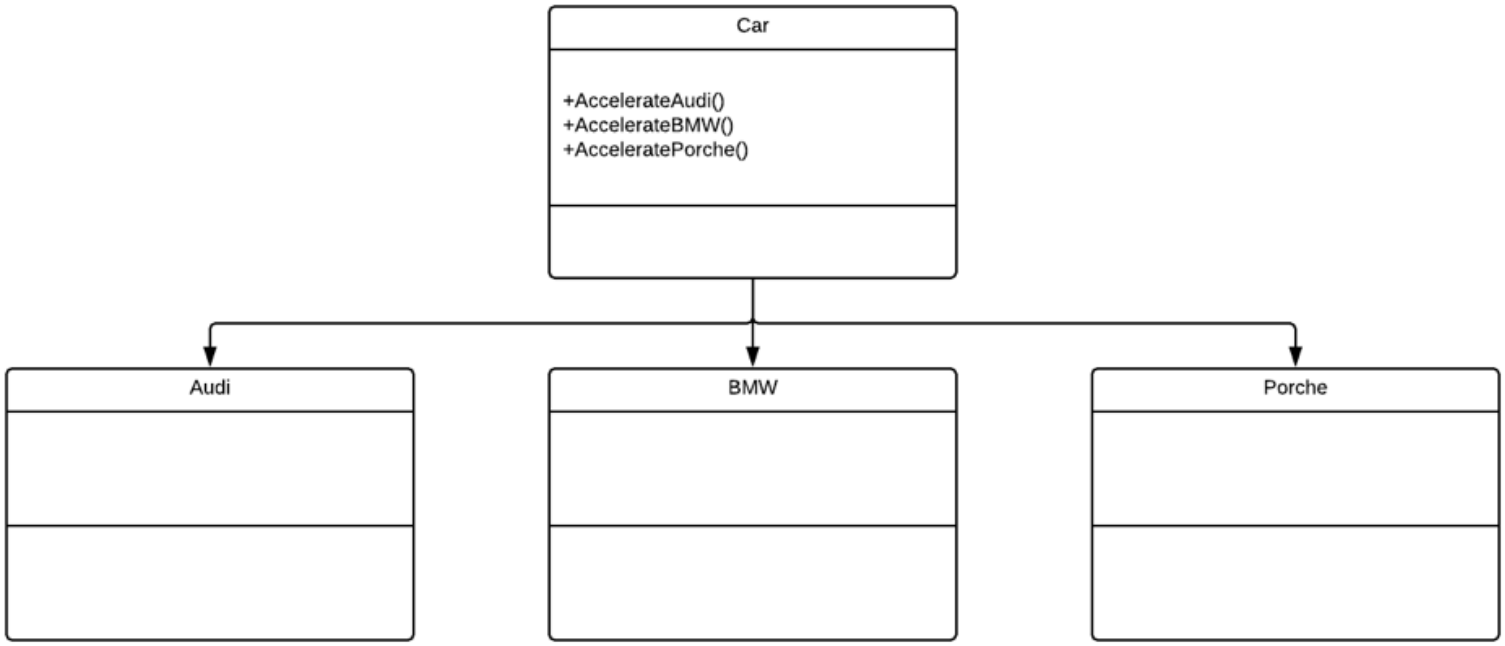
- Još jedan primjer:

```
public class HelloWorldAction extends Action
{
    @Override
    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
                                 throws Exception
    {

        //Process the request

    }
}
```

Izvor: https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/

Open Closed Principle

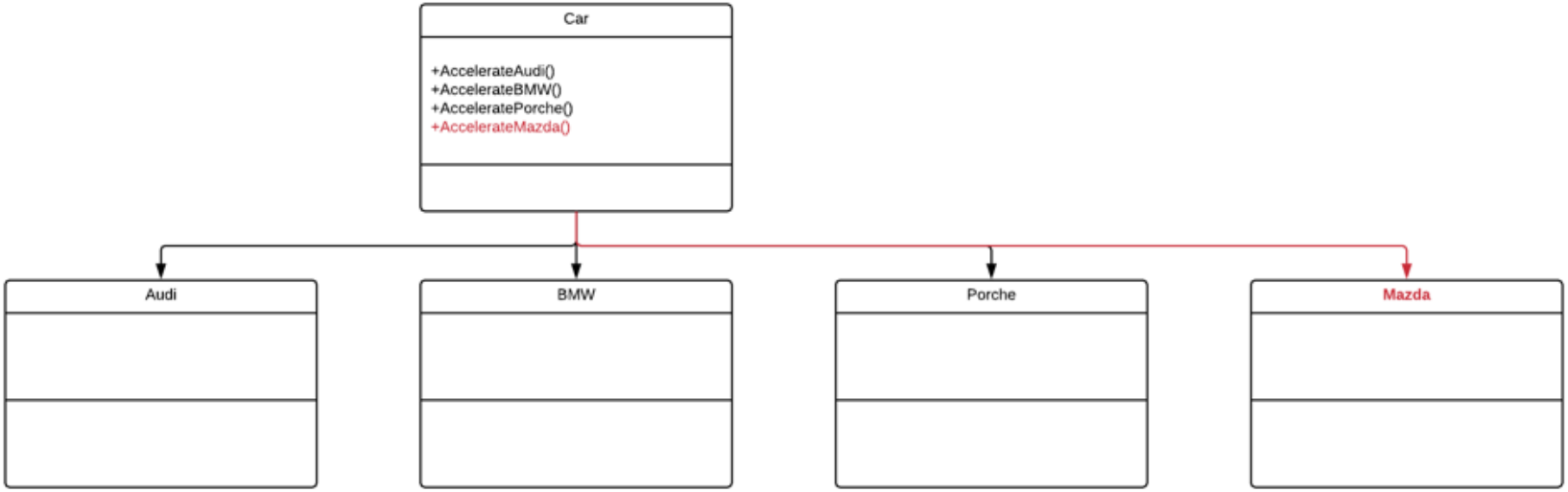
- Još jedan primjer:



Izvor: https://www.pluralsight.com/guides/solid-design-microservices

Open Closed Principle

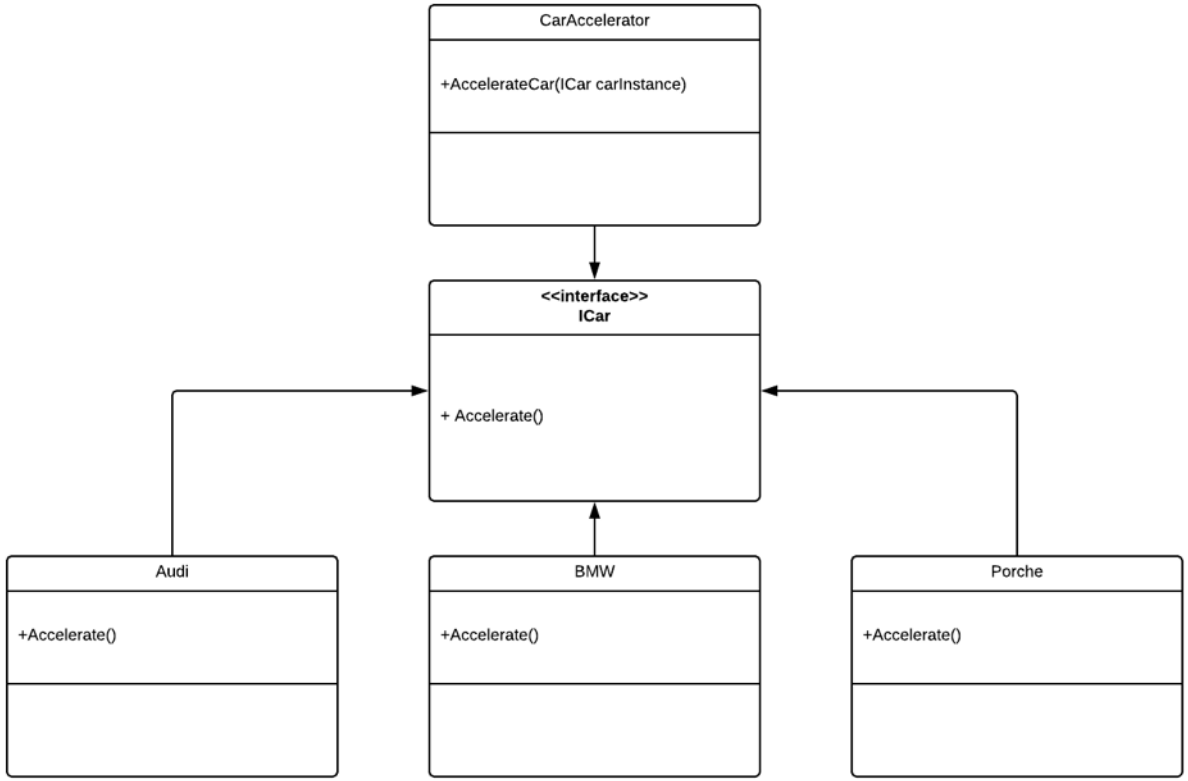
- Još jedan primjer:



Izvor: https://www.pluralsight.com/guides/solid-design-microservices

Open Closed Principle

- Još jedan primjer:



Izvor: https://www.pluralsight.com/guides/solid-design-microservices

Liskov Substitution Principle

- *"Funkcije koje koriste pokazivače ili reference na osnovne klase moraju moći koristiti objekte izvedenih klasa, a da to ne znaju." — Robert Martin, LSP dokument povezan s The Principles of OOD*

- Podklase bi se trebale lijepo ponašati kada se koriste umjesto njihove osnovne klase.

"Derived types must be completely substitutable for their base types"

Liskov Substitution Principle

```
// Prekršen Liskov Substitution Principle
 class Rectangle
 {
          int m_width;
          int m_height;

          public void setWidth(int width){
                    m_width = width;
          }

          public void setHeight(int h){
                    m_height = ht;
          }


          public int getWidth(){
                    return m_width;
          }

          public int getHeight(){
                    return m_height;
          }

          public int getArea(){
          return m_width * m_height;
          }
 }
```

```
class Square extends Rectangle
 {
          public void setWidth(int width){
                    m_width = width;
                    m_height = width;
          }

          public void setHeight(int height){
                    m_width = height;
                    m_height = height;
          }
 }
```

Izvor: https://www.oodesign.com/design-principles.html

Liskov Substitution Principle

```
class LspTest
{
private static Rectangle getNewRectangle()
{
        // to može biti objekt koji vraća neki factory...
        return new Square();
}

public static void main (String args[])
{
        Rectangle r = LspTest.getNewRectangle();
        r.setWidth(5);
        r.setHeight(10);
// korisnik zna da je r kvadrat, a očekuje se da može postaviti širinu i visinu

        System.out.println(r.getArea());
        // now he's surprised to see that the area is 100 instead of 50.
}
}
```

- Kao što se vidi u primjeru, ako bi neka *tvornica* vratila kvadrat gdje očekujemo pravokutnik, očekivani rezultati ne bi bili točni.

Izvor: https://www.oodesign.com/design-principles.html

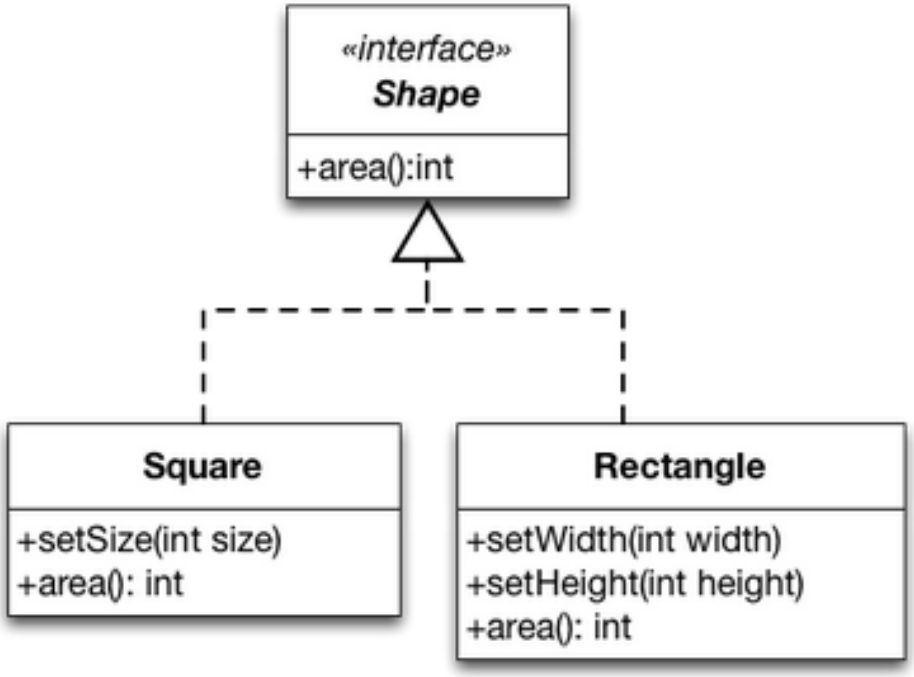
Liskov Substitution Principle

- Hijerarhija pravokutnika/kvadrata krši Liskovljev princip zamjene.
- **Ponašanje** kvadrata nije primjenjivo za pravokutnik
  - Kvadrat nije u skladu s ponašanjem pravokutnika: Promjena visine/širine kvadrata ponaša se drugačije od promjene visine/širine pravokutnika
  - Nema smisla razlikovati širinu i visinu kvadrata
- Ponašanje je srž softvera
  - Programeri ne definiraju entitete koji su nešto, već entitete koji se nekako ponašaju.
- Model pravokutnika/kvadrata promatran zasebno nije pokazao nikakve probleme. Valjanost modela ovisi o klijentima koji ga koriste. Moramo predvidjeti pretpostavke klijenata o našim klasama.

Izvor: Software Engineering Design & Construction , Software Technology Group Technische Universität Darmstadt

Liskov Substitution Principle

Rectangles and Square - LSP Compliant Solution



- Kada klijenti žele promijeniti svojstva oblika, moraju raditi s konkretnim klasama.
- Kada klijenti rade s konkretnim klasama, mogu točno pretpostaviti izračun površine.

Izvor: Software Engineering Design & Construction , Software Technology Group Technische Universität Darmstadt

Liskov Substitution Principle

- Još jedan primjer: svaka knjiga ima ISBN broj koji je uvijek u fiksnom formatu prikaza. Možete imati zasebne prikaze ISBN-a u bazi podataka i korisničkom sučelju. Za ovaj zahtjev, možemo napisati uređivač svojstava na takav način

```
public class IsbnEditor extends PropertyEditorSupport {
    @Override
    public void setAsText(String text) throws IllegalArgumentException {
        if (StringUtils.hasText(text)) {
            setValue(new Isbn(text.trim()));
        } else {
            setValue(null);
        }
    }

    @Override
    public String getAsText() {
        Isbn isbn = (Isbn) getValue();
        if (isbn != null) {
            return isbn.getIsbn();
        } else {
            return "";
        }
    }
}
```

Izvor: https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/

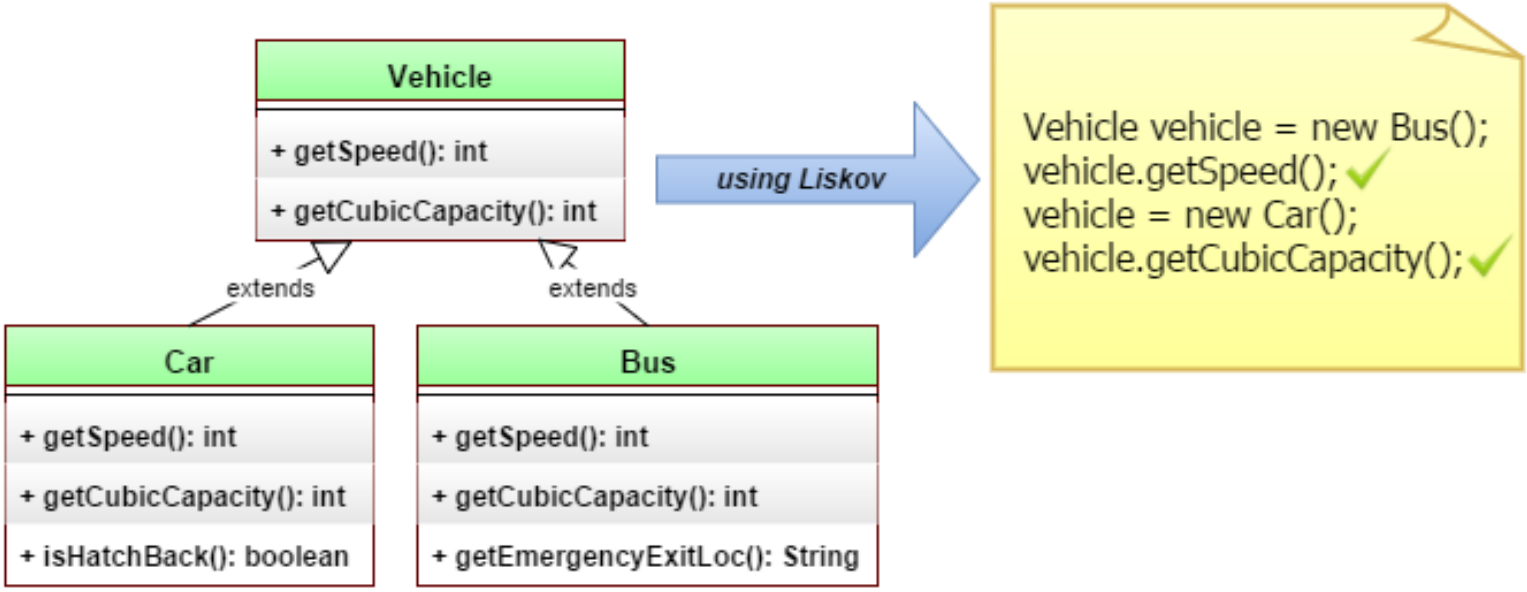
Liskov Substitution Principle

- Dakle, kada je potrebna registracija samo jednog uređivača svojstava za jednu vrstu podataka uz praćenje ograničenja koje nalaže osnovna klasa PropertyEditorSupport, svaka klasa koja proširuje PropertyEditorSupport klasu trebala bi biti ispravna zamjena svugdje gdje je potrebna osnovna klasa.

Izvor: https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/

Liskov Substitution Principle

- Još jedan primjer:



Izvor: https://www.javabrahman.com/programming-principles/liskov-substitution-principal-java-example/

Interface Segregation Principle

- *"Klijenti ne bi trebali biti prisiljeni ovisiti o sučeljima koja ne koriste." — Robert Martin, ISP dokument povezan s The Principles of OOD*
- Držite sučelja malenima

"Clients should not be forced to implement unnecessary methods which they will not use"

Interface Segregation Principle

- Nemojte prisiljavati klase da implementiraju metode koje ne mogu
- Nemojte zagađivati sučelja s puno metoda
- Izbjegavajte ‘debela' sučelja

Interface Segregation Principle

```
//loš primjer (zagađeno sučelje)
interface Worker {
 void work();
 void eat();
}
```

```
ManWorker implements Worker {
        void work() {…};
        void eat() {30 min
 break;};
}
```

```
RobotWorker implements Worker {
        void work() {…};
        void eat() {//Not Appliciable
                for a RobotWorker};
}
```

Izvor: Enbohm A. Solid – OO Design Principles

Interface Segregation Principle

- Rješenje
  - podijeljeno na dva sučelja

```
interface Workable {
        public void work();
}

interface Feedable{
        public void eat();
}
```

Izvor: Enbohm A. Solid – OO Design Principles

Interface Segregation Principle

- Još jedan primjer: rukovatelji događajima za rukovanje GUI događajima pokrenutim s tipkovnice i miša. Ima različite klase slušatelja za svaku vrstu događaja. Trebamo samo napisati rukovatelje za događaje kojima želimo upravljati. Ništa nije obavezno.

- Neki od slušatelja su:
  - `FocusListener`
  - `KeyListener`
  - `MosueMotionListener`
  - `MouseWheelListener`
  - `TextListener`
  - `WindowFocusListener`
- Uvijek kada želimo upravljati nekim od događaja, potrebno je odabrati željenog slušatelja i implementirati ga.

Izvor: https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/

Interface Segregation Principle

- Još jedan primjer: rukovatelji događajima za rukovanje GUI događajima pokrenutim s tipkovnice i miša. Ima različite klase slušatelja za svaku vrstu događaja. Trebamo samo napisati rukovatelje za događaje kojima želimo upravljati. Ništa nije obavezno.

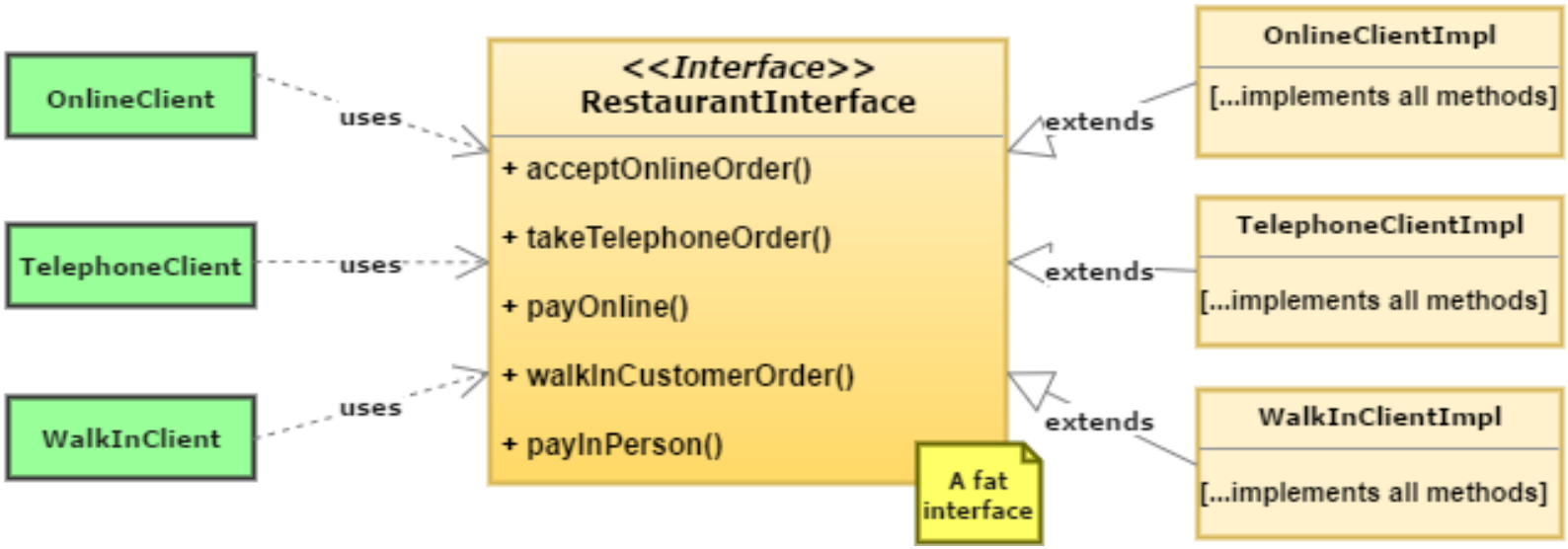
MouseMotionListenerImpl.java

```
public class MouseMotionListenerImpl implements MouseMotionListener
{
    @Override
    public void mouseDragged(MouseEvent e) {
        //handler code
    }

    @Override
    public void mouseMoved(MouseEvent e) {
        //handler code
    }
}
```

Izvor: https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/

Interface Segregation Principle

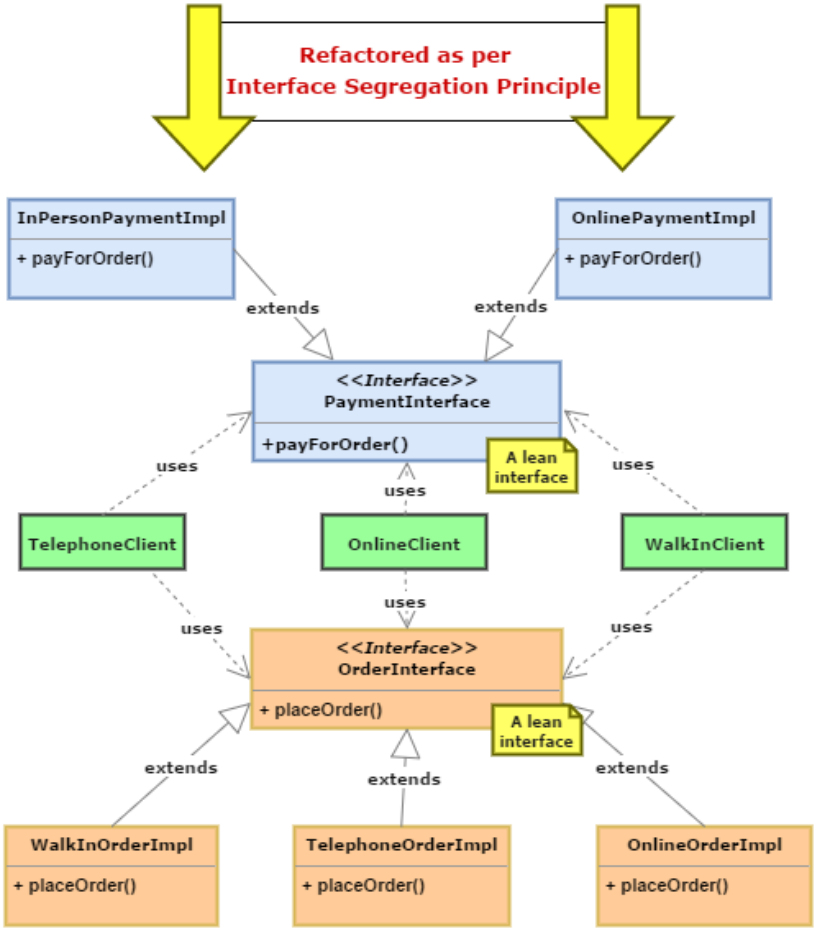
- Treći primjer:



Izvor: https://www.javabrahman.com/programming-principles/interface-segregation-principle-explained-examples-java/

Interface Segregation Principle

- Treći primjer:



Izvor: https://www.javabrahman.com/programming-principles/interface-segregation-principle-explained-examples-java/

Dependency Inversion Principle

- *"A. Moduli više razine ne bi trebali ovisiti o modulima niže razine. I jedni i drugi bi trebala ovisiti o apstrakcijama.*
  B. Apstrakcije ne bi trebale ovisiti o detaljima. Detalji bi trebali ovisiti o apstrakcijama." — Robert Martin, DIP rad povezan s The Principles of OOD

- Koristite puno sučelja i apstrakcija

> "Depend on abstractions, not on concretions"

Dependency Inversion Principle

```
//DIP – loš primjer
public class EmployeeService {
        private EmployeeFinder emFinder //concrete class, not abstract. Can access a SQL DB for instance
        public Employee findEmployee(…) {
                emFinder.findEmployee(…)
        }
}
```

Izvor: Enbohm A. Solid – OO Design Principles

Dependency Inversion Principle

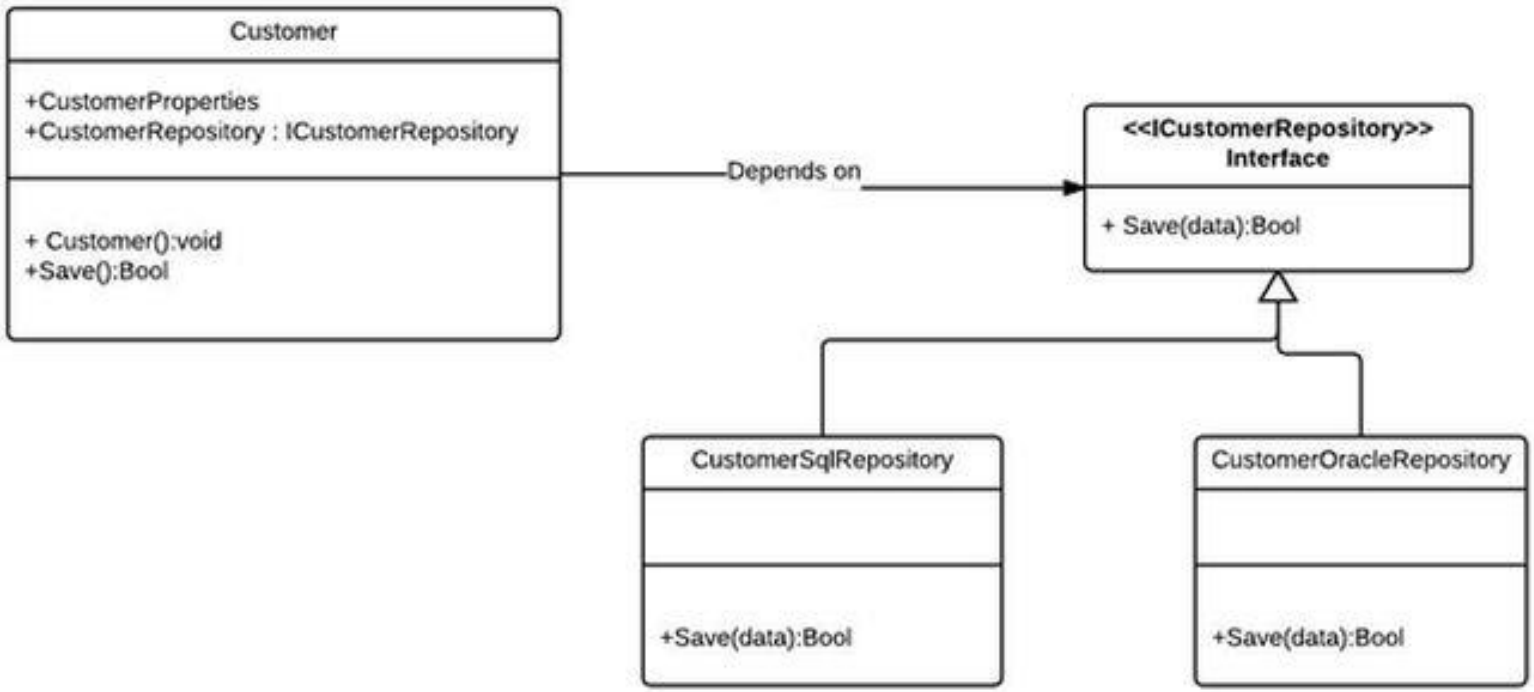
```
//DIP - fixed
public class EmployeeService {
        private IEmployeeFinder emFinder //depends on an abstraction, no an implementation
        public Employee findEmployee(…) {
                emFinder.findEmployee(…)
        }
}
```

- Sada je moguće promijeniti tražilicu tako da bude `XmEmployeeFinder`, `DBEmployeeFinder`, `FlatFileEmployeeFinder`, `MockEmployeeFinder`….

Izvor: Enbohm A. Solid – OO Design Principles

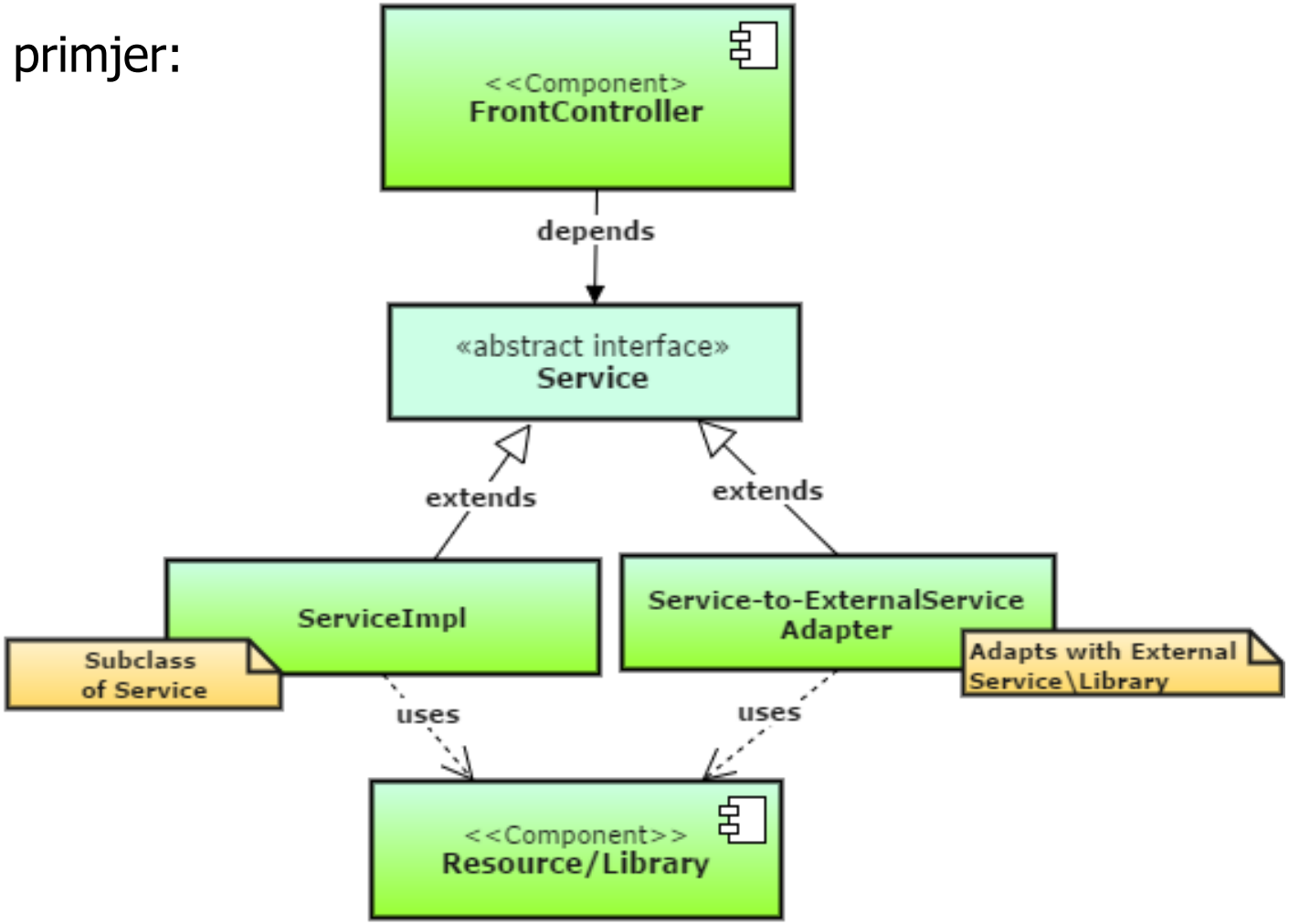
Dependency Inversion Principle

- Drugi primjer



Dependency Inversion Principle

- Treći primjer:



Izvor: https://www.javabrahman.com/programming-principles/dependency-inversion-principle-example-java/

Dependency Inversion Principle

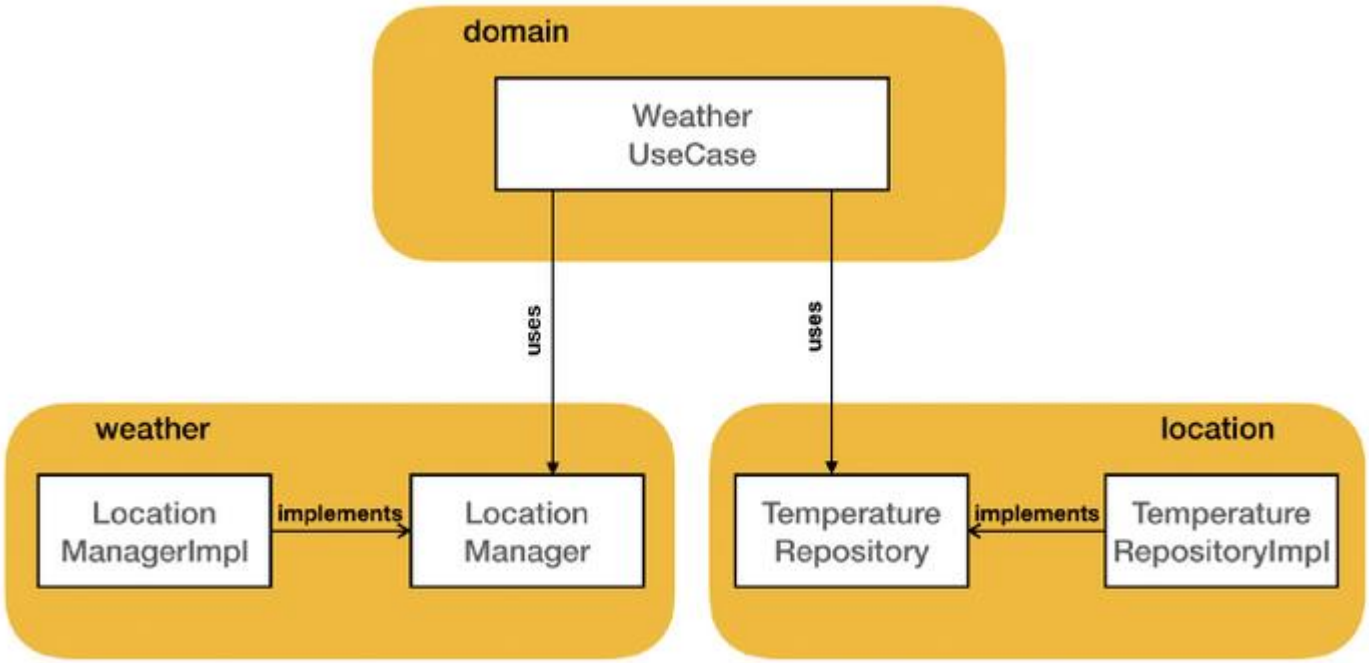
- Četvrti primjer:

```
class WeatherUseCase(
    private val locationManager: LocationManager,
    private val repository: TemperatureRepository) {

  suspend fun getCityData(): String = coroutineScope {
    try {
      val location = locationManager.getLastLocation()

      val cities = async { locationManager.getCities(location) }

      val temperature = repository.getTemperature(
            location.lat, location.lon)

      val city = cities.await().getOrElse(0) { "No city found" }
      "$city \n $temperature"
    } catch (e: Exception) {
      "Error retrieving data: ${e.message}"
    }
  }
}
```

Izvor: https://medium.com/google-developer-experts/implementing-dependency-inversion-using-dagger-components-d6b0fb3b6b5e

Dependency Inversion Principle

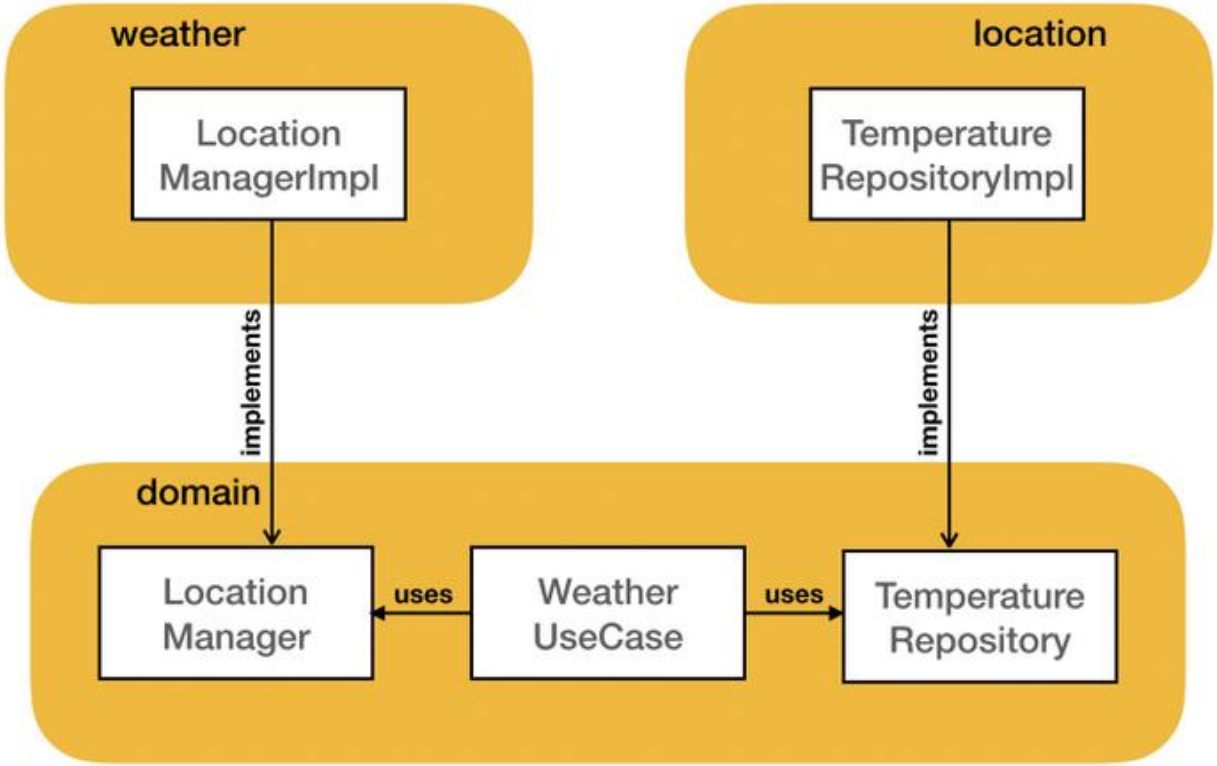
- Četvrti primjer:



Izvor: https://medium.com/google-developer-experts/implementing-dependency-inversion-using-dagger-components-d6b0fb3b6b5e

Dependency Inversion Principle

- Četvrti primjer:



Izvor: https://medium.com/google-developer-experts/implementing-dependency-inversion-using-dagger-components-d6b0fb3b6b5e

Zaključak

Single Resposibility Principle
A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)

Open / Closed Principle
A software module (it can be a class or method) should be open for extension but closed for modification.

Liskov Substitution Principle
Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

Interface Segregation Principle
Clients should not be forced to depend upon the interfaces that they do not use.

Dependency Inversion Principle
Program to an interface, not to an implementation.

Izvor: https://www.dckap.com/blog/why-should-every-developer-follow-solid-principles/

Zaključak

- Kod postaje *lakši za testiranje* (zapamtite da TDD nije samo testiranje, važniji je dizajn)
- Razvijajte softver *pametno*
  - ne provodite aktivnosti dizajna i razvoja „reda radi”
  - vrlo je važno vidjeti kontekst programa/kôda prilikom primjene SOLID-a
  - *Joel On Software* savjetuje – koristite se zdravim razumom!

Izvor: Enbohm A. Solid – OO Design Principles

Korišteni izvori

[1] butUncleBob.com, The Principles of OOP. Dostupno na: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[2] oodesign.com, Design Principles. Dostupno na: https://www.oodesign.com/design-principles.html

[3] Javabrahman.com, Programming & Design Principles. Dostupno na: https://www.javabrahman.com/programming-principles/

[4] Hikri K., SOLID Software Design Principles and How Fit in a Microservices Architecture Design. PluralSight.com, 2018., Dostupno na: https://www.pluralsight.com/guides/solid-design-microservices

[5] Devopedia. 2022. "SOLID Design Principles." Dostupno na https://devopedia.org/solid-design-principles

[6] Jayakanth R., 2018., Why should every Magento developer follow SOLID Principles?, Dckap.com, Dostupno na: https://www.dckap.com/blog/why-should-every-developer-follow-solid-principles/

[7] Gupta L., 2022. SOLID Principles. HowToDoInJava.com, Dostupno na: https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/

[8] Enbohm A. – SOLID – OO Design Principles, 2011. Dostupno na: www.slideshare.net/enbohm

[9] Collini F., 2019. Implementing Dependency Inversion using Dagger components, Medium.com, Dostupno na: https://medium.com/google-developer-experts/implementing-dependency-inversion-using-dagger-components-d6b0fb3b6b5e