SOLID LABORATEGIA

EGILEAK: Oier Elola eta Unax Lazkanotegi

OHARRA: Aurkibideko zenbaki bakoitzak galdera bakoitzari dagokio, galderak idatzita agertzen dira, baina dokumentu originala ikusi nahi izanez gero GitHub-eko errepositorioan bertan kopia bat utziko da ikusi ahal izateko (bere izena elab5-Solid.pdf da).

AURKIBIDEA

OCP

Zer gertatuko litzateke Sheet klaseari “Diamond” gehituko bagenio? Sheet klaseak OCP printzipioa betetzen du?. Erantzuna justifikatu. Betetzen ez badu, behar diren aldaketak egin printzipioa betetzeko.

Sheet klaseari “Diamond” gehituko bagenio ez luke OCP printzipioa betetzen; izan ere, Figure motako objektu bat izango litzateke eta Sheet klasean aldaketak egin beharko lirake bera sartzeko. Horretarako egindako aldaketak honakoak izan dira: Figure klasea sortu dugu, eta Diamond, Circle eta Square klaseek Figure klaseak dituen metodoak moldatuko dituzte klase bakoitzaren beharretara moldatuz.

```
public class Figure {
	public void draw(){
		System.out.println("Marraztu da "+ getClass());
	}
}

public class Square extends Figure {

	float length;
}

public class Circle extends Figure {

	float diameter;
}

public class Diamond extends Figure {

}


import java.util.Enumeration;
import java.util.Vector;

public class Sheet {
	Vector<Figure> figures=new Vector<>();

	public void addFigure(Figure s) {
		figures.add(s);
	}

	public void drawFigures() {
		Enumeration<Figure> efigures=figures.elements();
		Figure f;
		while (efigures.hasMoreElements()) {
			f=efigures.nextElement();
			f.draw();
		}
	}

}
```

Aplikazio bat sortu behar diren klase guztiekin, eta programa nagusi batean, drawFigures() metodoari deitzen dion programa bat sortu, bi circle, square bat eta diamond objektu batekin.

Jarraian ikusid dezakezue programa nagusiaren kodea eta funtzionatzen duela ikusteko kontsolak inprimatzen duena:

```
public class Test {
	public static void main(String[] args) {
		Square a= new Square();
		Circle b= new Circle();
		Circle c=new Circle();
		Diamond d=new Diamond();

		a.length=2;
		b.diameter=1;
		c.diameter=(float)0.5;

		Sheet s=new Sheet();
		s.addFigure(a);
		s.addFigure(b);
		s.addFigure(c);
		s.addFigure(d);

		s.drawFigures();
	}
}
```

Egin duzun aldaketa errefaktorizazio bat dela kontsideratzen duzu? Justifikatu erantzuna.

Errefaktorizazio printzipioak oinarritzat harturik, esan daiteke errefaktorizazio baten aurrean gaudela; izan ere, zenbait klaseetatik (kasu honetan, Square eta Circle dira hasieran zeuden klaseak) amankomunean dauden metodoak (draw() funtzioa) goi-klase berri batera (Figure klasera) eraman baititugu.

Figure bakoitzak azalera bat edukiko balu (hau da, getArea()), eta Sheet batean dauden irudi guztien azalera kalkulatu nahi bagenu, nola osatuko zenuke klaseak. Programa nagusia osatu, eskakizun hau frogatzeko.

getArea() metodoa Figure klasean kodetu dugu lehenengo modu honetara:

```
public float getArea() { return 0;}
```

Ondoren, klase hau jarraitzen duten Square eta Circle klaseetan getArea metodoa gainidatzi dugu:

- Square klasean

```
@Override
public float getArea() {
      return (length*length);
}
```

- Circle klasean

```
@Override
public float getArea() {
      return (float)(Math.PI*Math.pow(diameter, 2));
}
```

Sheet klasean calculateAreas metodoa sortu dugu non Figure klaseko objektu bakoitzari getArea() metodoari dei egiteko.

```
public void calculateAreas() {
      Enumeration<Figure> efigures=figures.elements();
      Figure f;
      while (efigures.hasMoreElements()) {
            f=efigures.nextElement();
            System.out.println("Area: " + f.getArea());
      }
}
```

Azkenik, test klasean `s.getArea()` metodoari dei egingo diogu ziurtatzeko ongi funtzionatzen duela.

OHARRA: Kodeketa guztia OCP paketean aurkitzen da.

SRP

Aplikazioa errefaktoriazatu, erresponsabilitate bakoitza klase isolatu batean geratu dadin. Adierazi zein aldaketa egin beharko litzateke fakturaren dedukzioa (hau da, `billDeduction`) fakturaren arabera kalkulatuko balitz:

```
if (billAmount >50000)
        billDeduction = (billAmount * deductionPercentage +5) / 100;
else
        bill Deduction = (billAmount * deductionPercentage) / 100;
```

Hiru erresponsabilitate nagusi daudela ikusi daiteke kodean, eta SRP printzipioak dioen bezala, klase batek erresponsabilitate bakarra izan behar du. Beraz, dedukzioa kalkulatzeko eta VAT balioa kalkulatzeko klase bana sortuko ditugu calculoDeduction eta calculoVAT izenekoak, hurrenez hurren, horrela Bill klaseak SRP printzipioa beteko du.

Jarraian, klase bakoitzaren kodea eta Bill klaseko totalCalc() metodoa nola kodetuta geratu diren ikusi daiteke.

```
public class calculoDeduction {
        public float calcDeduction(float billAmount,float deductionPercentage) {
                return (billAmount*deductionPercentage)/100;
}

public class calculoVAT {
        public float calcVAT(float billAmount,float percentage) {
                return (float) (billAmount * percentage);
        }
}

public void totalCalc() {
        calculoVAT b=new calculoVAT();
        calculoDeduction c=new calculoDeduction();
        billDeduction=c.calcDeduction(billAmount, deductionPercentage);
        VAT =b.calcVAT(billAmount,(float)0.16);
        billTotal= (billAmount-billDeduction) + VAT;
}
```

Fakturaren dedukzioa fakturaren arabera kalkulatuko balitz, egin beharko litzatekeen aldaketa calculoDeduction klaseko calcDeduction metodoan aldatu beharko genuke, eta if-else egituraz kontrolatu billAmount kantitatea. Azken kodeketa honakoa da:

```
public float calcDeduction(float billAmount,float deductionPercentage) {
        if (billAmount>5000) {
                return (billAmount*deductionPercentage + 5 ) / 100;
        }else {
                return (billAmount * deductionPercentage) / 100;
        }
}
```

Zein aldaketa egin beharko litzateke VAT-a %16-tik %18-ra aldatuko balitz?

Kasu honetan, Bill klasean egingo litzateke aldaketa txiki hori; izan ere, erresponsabilitate hau burutzen duen calculoVAT klasean kodetuta dagoen calcVAT funtzioak bi parametro jasotzen ditu: alde batetik billAmount balioa eta bestetik portzentaia, beraz, erabiltzaileak edozein momentutan alda dezake portzentaiaren balioa totalCalc() funtzioan bertan.

Zein aldaketa egin beharko litzateke, 0 kodea duten fakturei VAT-a aplikatzen EZ bazaie

Erresponsibilate hau hiruetan azkena izango litzateke, totala kalkulatzerako unean kontutan hartu beharko litzateke ea kodea 0 den ala ez. Beraz, if-else egitura baten bidez kontrolatuko dugu ea fakturaren kodea zein den, eta horren arabera aplikatuko da VAT-a ala ez.

```
	if (code.compareTo("0")==0) {
		billTotal = (billAmount - billDeduction) + VAT;
	}else {
		billTotal= (billAmount-billDeduction);
	}
```

Jarraian ikusi dezakezue kodea nola geratu den amaieran, aurretik egindako aldaketak barne direlarik.

```
public void totalCalc() {
	calculoVAT b=new calculoVAT();
	calculoDeduction c=new calculoDeduction();
// Dedukzioa kalkulatu
	billDeduction=c.calcDeduction(billAmount, deductionPercentage);
// VAT kalkulatzen dugu
	VAT =b.calcVAT(billAmount,(float)0.18);
// Totala kalkulatzen dugu
	if (code.compareTo("0")==0) {
		billTotal = (billAmount - billDeduction) + VAT;
	}else {
		billTotal= (billAmount-billDeduction);
	}
}
```

OHARRA: Kodeketa guztia SRP klasean aurkitzen da.

DIP

DIP printzipioa betetzen da? Zer aldatu behar da?

Student klasea daukagu register izeneko metodo bati dei egiten, baina bertan, hiru mota ezberdineko azpiklaseak erabiltzen ditu zuzenean ezagutuz. Orduan, horrek esan nahi du DIP printzipioa ez duela betetzen.

Errefaktorizazioa

Programa honek DIP printzipioa bete dezan, lehenengo Student klaseak erabiltzen dituen azpiklaseei *interfaze/abstrakzio* batzuk gehitu behar dizkiegu eta horretarako, lehenengo *interfazeak/klase abstraktuak* sortu behar ditugu, kasu honetan interfazeak:

- Deduction klasearen interfazea

```
public interface DeductionAbstraction {
	int calcDeduction(String sex, String year);
}
```

- Precondition klasearen interfazea

```
import java.util.HashMap;
public interface PreconditionsAbstraction {
	boolean isPossible(String subject, HashMap<String, Integer> records);

}
```

- SubjectQuotes klasearen interfazea

```
public interface SubjectQuotesAbstraction {
	int getPrice(String subject);
}
```

Eta behin aldaketa hauek eginda, Deduction, Preconditions eta SubjectQuotes klaseak zuzenduko ditugu:

```
public class Deduction implements DeductionAbstraction{
...
}
```

```
import java.util.HashMap;
public class Preconditions implements PreconditionsAbstraction{
...
}
```

```
public class SubjectQuotes implements SubjectQuotesAbstraction {
• • •
}
```

Honen ondoren geratzen den pausu bakarra, Student klasean aldaketa txiki bat egitea da, hasieran batean sortzen zituen objetuak(Preconditions, Deduction eta SubjectQuotes motakoak) parametro bidez sartzea zuzenean eta honela klase hauei zuzenean dei egin behar izatea eragozteko:

```
import java.util.HashMap;
public class Student {
• • •
// Irakasgai batean matrikulatzen duen metodoa.
	public void register(String subject, PreconditionsAbstraction p,
				DeductionAbstraction d,SubjectQuotesAbstraction sq) {
// Aurrebaldintzak konprobatzen dira
		boolean isPosible = p.isPossible(subject, subjectRecord);
		if (isPosible) {
			// Dedukzioa kalkulatu sex eta edadearen arabera
			int percentage = d.calcDeduction(sex, year);
// Irakasgaiaren prezioa lortu
			int quote = sq.getPrice(subject);
// HashMap batean gordetzen du eta ordaindu behar duen balioa eguneratu
			subjectRecord.put(subject, null);
			toCharge = toCharge + (quote - percentage * quote / 100);
		}
	}
}
```

Aldaketa hauekin, programa DIP printzipioa betetzen du eta ez du beharrik azpiklaseak zuzenean atzitzeko.

OHARRA: Kodeketa guztia DIP paketean aurkitzen da.

LSK(Bukatzeko)

Programa batean, Project bat sortu 3 fitxategi mota desberdin batzuekin, eta jarraian bere metodoak exekutatu.

Jarraian ikusi daitekeena izango litzateke programa: Project bat sortu dugu eta 3 fitxategi mota (bi ProjectFile eta ReadOnlyProjectFile bat) gehitu ditugu bertara eta metodoak exekutatu.

Project klaseak OCP printzipioa betetzen du? Justifikatu erantzuna.

Bai, OCP printzipioa betetzen da; izan ere, ProjectFile klaseak ahalbidetzen digu zehaztasun handiagoko ProjectFile objektuak sortzeko (adibidez, ReadOnlyProjectFile).

Project klaseak LSK (Liskov) printzipioa betetzen du? Justifikatu erantzuna.

LSK printzipioa ez da betetzen; izan ere, ReadOnlyProjectFile klaseko objektu batek, izenak dioen bezala, ezin du fitxategia gorde. Baina ProjectFile klasearen extensio bat denez, store() funtzioa heredatzen du non errorea jaurtiko duen, eta ez da desiragarria hori gertatzea.

Errefaktorizatu aplikazioa OCP edo LSK betetzen ez badu.

Egin den aldaketa nagusia honakoa da: Readable eta Storable interfazeak sortu dira, eta bakoitzak honako kodeketa jarraitzen du:

```
public interface Readable{
	public void read();
}

public interface Storable{
	public void store();
}
```

Noski, orain ProjectFile eta ReadOnlyProjectFile klaseek, bakoitzak behar dituen interfaze horiek inplementatuko ditu: ProjectFile-n kasuan biak eta ReadOnlyProjectFile-k soilik Readable

```
public class ReadOnlyProjectFile extends File implements Readable  {
	public ReadOnlyProjectFile(String filePath) {
		super(filePath);
	}

	public void read() {
		System.out.println("file loaded from " + filePath);
	}
}
```

```
public class ProjectFile extends File implements Readable,Storable{
	public ProjectFile(String filePath){
		super(filePath);
	}
	public void read() {
		System.out.println("file loaded from " + filePath);
	}
	public void store() {
		System.out.println("file saved to " + filePath);
	}
}
```

Aldaketa ordea, File klasearen sorrera da: izan ere, ReadOnlyProjectFile klaseak nahiz eta soilik inplementatu, ProjectFile klasearen metodoak inplementatzen dituen, store() egingo luke, eta beraz File klasea sortu dugu:

```
public class File implements Readable,Storable{
	public String filePath;
	public File(String filePath) {
		this.filePath = filePath;
	}
	public void read() {
		System.out.println("file loaded from " + filePath);
	}
	public void store() {
		System.out.println("file saved to " + filePath);
	}
}
```

Proba programan aldaketarik ez da egin behar izan, eta orain LSK printzipioa betetzen dela zihurta dezakegu.

OHARRA:Kodeketa guztia LSK klasean aurkitzen da.

ISP

Betetzen da ISP printzipioa? Zer behar dute EmailSender eta SMSSender klaseek?

EmailSender klaseak Pertson motako objetuaren emaila behar du eta SMSSender klaseak aldiz Pertson motako objetuaren telefonoa. Baina biek Pertson motako objetua zuzenean atzitzen dute eta horregatik beharko ez dituzten datuak atzitu ditzakete, adibidez, EmailSender klasearen kasuan telefono zenbakia ez da beharrezkoa, baina atzitu dezake. Horrek esan nahi du ISP printzipioa bortxatu egiten dutela, beharrezkoa behar ez dituzten datuak baino gehiago atzitzeko aukera dutelako.

Errefaktorizazioa

Lehenengo, bi interfaze sortuko ditugu, Pertson klaseak inplementatu ahal izateko eta horrela kasu bakoitzean interfazeak eskaintzen dituen datuak eskaini ahalko dira:

- EmailSendable klasea, EmailSender klasearentzat:

```
public interface EmailSendable {
	String getEmail();
}
```

- Telephonable klasea, SMSSender klasearentzat:

```
public interface Telephonable {
	String getTelephone();
}
```

Aldaketa hauek egin ondoren, Pertson klaseak inplementatzea egin beharko dugu:

```
public class Person implements EmailSendable, Telephonable {
	String name, address, email, telephone;
...
}
```

Azkenik, aldaketak amaitzeko EmailSender eta SMSSender klaseetan egin beharko ditugu, metodoen deiak interfazeei eginik zuzenean Pertson klaseari atzitu gabe:

- EmailSender klasea:

```
public class EmailSender {

	public void sendEmail(EmailSendable c, String message) {
		// Mezu bat bidaltzen du Person klaseko korreo helbidera.
	}
}
```

- SMSSender klasea:

```
public class SMSSender {
	public static void sendSMS(Telephonable c, String message) {
		// SMS bat bidaltzen du Person klaseko telefono zenbakira.
	}
}
```

Gehitutako beste mail klasea

Gainera, EmailSendable klasea inplementatzen duen beste klase bat sortu dugu, EmailSenderrak honekin interaktuatzeko aukera izan dezan GmailAccount klasea:

```
public class GmailAccount implements EmailSendable {
	String name, emailAddress;

	public GmailAccount(String izena, String helbidea) {
		this.name = izena;
		this.emailAddress = helbidea;
	}

	public String getEmail() {
		return emailAddress;
	}
}
```

Emaitza proban

Egindako programaren proba hurrengoa da, Main klasean dagoena:

```
public class Main {
	public static void main(String[] args) {
		Person p = new Person("Unax", "Amute", "ulazkanotegi010@ikasle.ehu.eus",
"688825012");
		String mezua = "Hello World!!!";
		EmailSender es = new EmailSender();
		es.sendEmail(p, mezua);
		System.out.println(p.getName() + " pertsonari email mezua bidali zaio: " +
mezua);

		SMSSender.sendSMS(p, mezua);
		System.out.println(p.getName() + " pertsonari SMS mezua bidali zaio: " +
mezua);

		GmailAccount ga = new GmailAccount("Unax","unacen@gmail.com");
		es.sendEmail(ga, mezua);
		System.out.println(ga.getEmail() + " emailari mezua bidali zaio: " + mezua);
	}
}
```

OHARRA:Kodeketa guztia DIP klasean aurkitzen da.

GitHub-era esteka

Atal honetan aurkitu dezakezue GitHub-eko errepositorioaren link-a, non bertan aurkitu dezakezue Java proiektua eta fitxategi hau bera ere.

GitHub-eko esteka: https://github.com/Elola27/Solid_ElolaLazkanotegi