1. S.O.L.I.D.

1.1.SINGLE RESPONSABILITY PRINCIPLE

O clasa trebuie sa aiba intotdeauna o singura responsabilitate si numai una. In caz contrar, orice schimbare de specificatii va duce la inutilitatea ei si rescrierea intregului cod.

Ex1: Implementarea de mai jos incalca principiul **S**, deoarece **NetIncomeCalculator** stie sa faca doua lucruri diferite, si anume: sa calculeze netul total si sa afiseze la consola. Aceasta clasa devine inutila in momentul in care dorim sa adaugam o alta metoda pentru afisarea rezultatul, cum ar fi *displayInFile():*

```
public class NetIncomeCalculator implements NetIncomeCalculatorInterface {

	private Map<Integer, Person> persons;

	public NetIncomeCalculator(Map<Integer, Person> persons) {
		this.persons = persons;
	}

	@Override
	public double calculateTotalNetIncome() throws NotFoundPersonException {

		//implementare
	}

	public void displayConsole(NetIncomeCalculatorInterface calculator) throws
NotFoundPersonException {
		//implementare
	}
}
```

Solutie: Se defineste o alta clasa, **Report**, care sa contina toate metodele pentru afisarea informatiilor din **NetIncomeCalculator**. Iar aceasta din urma, o sa implementeze, doar acele metode care prelucreaza datele incapsulate in obiectele de tip **Person.**

1.2.OPEN-CLOSED PRINCIPLE

Obiectele sau entitatile trebuie sa fie deschise pentru extindere, dar inchise pentru modificari.

Ex2: In momentul compilarii metodei de mai jos nu stim tipul concret al obiectelor de tip **Person,** ceea ce a condus la utilizarea blocurile de tip if-then-else if pentru identificarea acestuia. Modul acesta de implementare incalca principiul **O,** deoarece in momentul in care o sa apara un alt tip concret al clasei **Person** (spre exemplu InginerAtom) metoda *calculateTotalNetIncome()* trebuie modificata pentru a putea calcula suma si pentru noul caz.

```
	public double calculateTotalNetIncome() throws NotFoundPersonException {

		if (persons == null || persons.size() == 0)
			throw new NotFoundPersonException("Not found persons");

		double sum = 0;

		for (Person person : persons.values()) {

			if (person instanceof Programmer) {
				sum += (1 - 0.13 - 0.2) * person.getGrossIncome();
			} else if (person instanceof Driver) {
				sum += (1 - 0.13 - 0.2 - 0.1) * person.getGrossIncome();
			} else if (person instanceof ScholarshipStudent) {
				sum += person.getGrossIncome();
			}
		}

		return sum;
	}
```

Solutie: Definirea unei metode, *calculateNetIncome()*, care poate sa fie abstracta sau concreta (se alege cel mai des intalnit comportament al functiei respective), in cadrul clasei **Person,** iar fiecare clasa concreta trebuie sa suprascrie aceasta metoda, cu comportamentul asteptat. O alta solutie o reprezinta definirea unei interfete care sa contina metoda descrisa anterior.
Forma finala a metodei *calculateTotalNetIncome()* ar trebui sa arate astfel:

```
	public double calculateTotalNetIncome() throws NotFoundPersonException {

		if (persons == null || persons.size() == 0)
			throw new NotFoundPersonException("Not found persons");

		double sum = 0;

		for (Person person : persons.values()) {
			sum += person.calculateNetIncome();
		}

		return sum;
	}
```

1.3.LISKOV SUBSTITUTION PRINCIPLE

Obiectele pot fi inlocuite oricand cu instante ale claselor derivate fara ca acest lucru sa afecteze functionalitatea.

Ex3: Clasa **ReportCsv** extinde **Report,** dar implementeaza doar metoda *displayInFile().*

```
public class ReportCsv implements Report {

	@Override
	public void displayConsole(NetIncomeCalculatorInterface calculator) throws
		NotFoundPersonException {
		throw new Exception("Not Implemented");
	}

	@Override
	public void displayInFile(NetIncomeCalculatorInterface calculator)
			throws IOException, NotFoundPersonException {

		File file = new File("Info.csv");
		BufferedWriter writer = new BufferedWriter(new FileWriter(file));

		for (Integer id : calculator.keys()) {

			writer.append(calculator.calculateNetIncomeById(id));
			writer.newLine();
		}

		writer.write("Net income of the family is: "
				+ String.valueOf(calculator.calculateTotalNetIncome()));
		writer.close();
	}

}
```

Conform **L**, urmatoarele linii de cod ar trebui sa ruleze fara erori de compilare sau executie:

```
 Report report = new Report();
 Report reportCsv = new ReportCsv();

 try {
	reportCsv.displayConsole(calculator);
	reportCsv.displayInFile(calculator);

} catch (NotFoundPersonException e) {
	e.printStackTrace();
} catch (Exception e) {
	e.printStackTrace();
}
```

Liniile de mai sus nu o sa functioneze, deoarece in implementarea metodei *displayConsole()*, din clasa **ReportCsv**, este aruncata o execeptie. Ceea ce duce la incalcarea principiului Liskov, avand in vedere ca instanta *reportCsv* de tip **Report,** nu poate fi inlocuita cu succes de catre implementarea derivatei acesteia, **ReportCsv.**

Solutie: Pentru rezolvarea acestui principiu trebuie ca parintele clasei **ReportCsv** sa fie inlocuit de catre o noua interfata, **ReportFileInterface,** care sa contina metoda *displayInFile().* Aceasta interfata, o sa fie implementata atat de **ReportCsv,** cat si de **Report** (aceasta clasa o sa contina si metoda *displayConsole()).* Prin urmare, forma corecta a acestui principiu se poate observa in urmatoarele linii de cod:

```
ReportFileInterface report = new Report();
ReportFileInterface reportCsv = new ReportCsv();

try {
	((Report) report).displayConsole(calculator);
	report.displayInFile(calculator);

	// reportCsv.displayConsole(calculator);// nu este implementat
	reportCsv.displayInFile(calculator);

} catch (NotFoundPersonException e) {
	e.printStackTrace();
} catch (IOException e) {
	e.printStackTrace();
}
```

1.4.INTERFACE SEGREGATION PRINCIPLE

Nu se defineste o singura interfata care sa contina toate metodele necesare.
Mai multe interfete specializate sunt oricand de preferat unei singure interfete generale.
Obiectele nu trebuiesc obligate sa implementeze metode care nu sunt necesare.

Ex4: Fie clasa **NetIncomeCalculator,** care contine o singura metoda, si anume *calculateTotalNetIncome().* De asemenea, clasa abstract **Person (atribute: name, age, grossIncome)** si derivatele acestia: **Driver(atribute: category), Programmer (atribute: position, level) si ScholarshipStudent(atribute: faculty, group, year).** Se doreste adaugarea unei noi metode *calculateTaxIncome(),* procentele pentru cele trei tipuri de date sunt: **ScholarshipStudent si Programmer 0%, iar Driver 10%**. Daca aceasta

metoda este adaugata in **NetIncomeCalculator** sau in **Person**, duce la incalcarea principiului **I:**

In primul caz, poate fi incalcat si principiul **O,** prin simplu fapt ca tipul concret al clasei este necunoscut la compilare (in plus, in momentul in care se doreste adaugarea unui nou tip va trebui sa modificam aceasta metoda):

```
public double calculateTaxIncome(Integer id) throws NotFoundPersonException {

	if (persons == null || persons.size() == 0 || id == null
			|| !persons.containsKey(id))
		throw new NotFoundPersonException("Not found persons");

	if (persons.get(id) instanceof Driver)
		return 0.1 * persons.get(id).getGrossIncome();

	return 0;
}
```

In al doilea caz, este incalcat principiul **I**, deoarece toate clasele concrete o sa aiba acest comportament desi, pentru clasele **ScholarshipStudent si Programmer** a caror valoare este 0 tot timpul, nu este necesar.

Solutie: Se defineste o noua interfata, **TaxIncomeInterface,** care contine metoda *calculateTaxIncome().* Aceasta este implementata doar de catre clasa **Driver.**

1.5.DEPENDENCY INVERSION PRINCIPLE

O clasa trebuie sa depinda de abstractizare, nu de obiecte concrete. In felul aceasta decuplarea unei implementari curente din solutie este foarte usor inlocuita cu una noua, fara a avea erori de compilare.

Ex5: Metoda de mai jos, primeste ca si parametru de intrare un obiect de tipul clasei **NetIncomeClaculator.** Principiul **D** este incalcat in implementarea urmatoare:

```
public void displayConsole(NetIncomeCalculator calculator) throws
	NotFoundPersonException {

	double sum = calculator.calculateTotalNetIncome();
	System.out.println("Net income of the family is: "
			+ String.valueOf(sum));
}
```

Solutie: Se defineste o noua interfata, **NetIncomeCalculatorInterface,** care contine toate metodele clasei **NetIncomeCalculator.** Aceasta din urma implementeaza noua interfata, iar metoda *displayConsole()*, o sa primeasca un obiect de tipul interfetei.

```
	public void displayConsole(NetIncomeCalculatorInterface calculator) throws
		NotFoundPersonException {
		double sum = calculator.calculateTotalNetIncome();
		System.out.println("Net income of the family is: "
				+ String.valueOf(sum));
	}
```

2. DON'T REPEAT YOURSELF (DRY)

Conform acestui principiu, niciodata nu trebuie sa se repete aceleasi linii de cod in cel putin doua metode.

```
public static void creditBank(Person person) {
	expressCreditBank(person);
	normalCreditBank(person);
}

public static boolean isAvailableCreditBank(Person person) {
	return person != null && (person.getAge() > 18 && person.getAge() < 66)
			&& person.getGrossIncome() != null
			&& person.getGrossIncome() > 0;
}

public static void expressCreditBank(Person person) {

	if (isAvailableCreditBank(person) && person.calculateNetIncome() > 5000)
	{
		System.out.println("You can make an express credit bank");
	} else {
		System.out.println("You can't make credit bank");
	}
}

public static void normalCreditBank(Person person) {

	if (isAvailableCreditBank(person)
			&& (person.calculateNetIncome() > 2000 && person
					.calculateNetIncome() <= 5000)) {
		System.out.println("You can make a normal credit bank");
	} else {
		System.out.println("You can't make credit bank");
```

```
	}
}
```

In codul de mai sus se poate observa ca metodele *expresCreditBank()* *si normalCreditBank()* au aproape aceeasi implementare, singura diferenta fiind a doua conditie din blocul if. Prin urmare, principiul DRY este incalcat.

Forma corecta a implementarii de mai sus, conform DRY, este:

```
public static void creditBank(Person person) {
	dryCreditBank(person);
}

public static boolean isAvailableCreditBank(Person person) {
	return person != null && (person.getAge() > 18 && person.getAge() < 66)
			&& person.getGrossIncome() != null
			&& person.getGrossIncome() > 0;
}



public static void dryCreditBank(Person person) {

	if (isAvailableCreditBank(person)) {
		if (person.calculateNetIncome() > 2000
				&& person.calculateNetIncome() <= 5000) {
			System.out.println("You can make a normal credit bank");
		} else {
			System.out.println("You can make an express credit bank");
		}
	} else {
		System.out.println("You can't make a credit bank");
	}
}
```

3. KEEP IT SIMPLE AND STUPID (KISS)

Niciodata nu trebuie sa scriem tot codul sursa intr-o singura metoda sau cu alte cuvinte, o metoda trebuie sa faca un singur lucru si numai unul.

4. YOU AIN'T GONNA NEED IT (YAGNI)

Nu scriem metode care nu sunt necesare inca, acestea ar putea ramane nefolosit.