Model de proiect privind aplicarea sabloanelor.

Nota! Proiectul trebuie sa contina cate un sablon de proiectare din fiecare dintre cele 3 categorii pentru fiecare student.

Nota! Exemplul de aplicare a unui sablon trebuie descris conform modelului de mai jos

Nota! Partea a 2-a a proiectului va fi prezentata la ultimul laborator din semestru.

Sablonului Strategy

1. **Definirea problemei.**

Se doreste implementarea a doua functionalitati in aplicatia pentru managementul proiectelor:

a) estimarea costurilor unui proiect, conform uneia dintre metodele: modelul COCOMO, metoda ABC, COSYSMO, Monte Carlo etc.
b) calculul costului efectiv al unui proiect, conform uneia dintre metodele: metoda direct costing, costul standard, pe comenzi, pe faze, etc.

Aceste metode pot fi stabilite la momentul crearii unui proiect nou, insa ele pot fi schimbate ulterior sau pot fi inlocuite cu altele nou aparute. O alta cerinta in acest sens consta in posibilitatea alegerii metodei aplicate de catre utilizator, situatie in care ea nu va fi cunoscuta decat la momentul executiei aplicatiei („run-time").

Pentru implementarea celor 2 cerinte se vor introduce 2 atribute in clasa Proiect, respectiv CostCalculat si CostEstimat.

Acesta fiind contextul, problema concreta este urmatoarea: cum vor fi implementate metodele de estimare si calcul al costului proiectelor, fiecare implicand algoritmi distincti de prelucrare?

2. **Solutii fara aplicarea unui sablon de proiectare**.

Pentru rezolvarea acestei probleme ar putea fi imaginate 2 solutii:

i. Implementarea in clasa Proiect a tuturor metodelor si modelelor de calcul, astfel:
   - se adauga 2 atribute pentru setarea pentru un proiect a metodelor folosite in cele 2 calcule: ModelEstimare si MetodaCalculCost (enumeration class). Cele 2 atribute vor stoca numele metodelor folosite pentru estimarea si calculul efectiv al costului, stabilite de catre utilizator in momentul crearii unui proiect sau modificate ulterior;
   - se adauga cate o operatie (metoda) pentru fiecare metoda de estimare/calcul cost, care va implementa algoritmul specific fiecarei metode;
   - se creeaza cate o operatie (metoda) pentru fiecare dintre cele doua categorii de metodele (estimare sau calcul cost) estimeazaCost() si calculCost(). Aceste metode vor implementa algoritmii de alegere a metodelor de estimare/calcul al costului, in functie de valorile celor 2 atribute ModelEstimare/MetodaCalculCost.

Codul de implementare a solutiei:

```
public enum ModelEstimare {
	COCOMO, COSYSMO, ABC, MonteCarlo
}

public enum MetodaCalculCost {
	DirectCosting, CostStandard, PeComenzi, PeFaze
}

public class Proiect {
	private Double buget;
	private ModelEstimare modelEstimare;
	private Double costEstimat;
	private MetodaCalculCost metodaCalculCost;
	private Double costCalculat;
	private Long id;
	private String titlu;

	// atribuirea valorilor celor doua attribute nu se face direct prin
	// specificarea unei valori ca parametru a metodei SET, ci prin
	// invocarea metodei corespunzatoare
	public Double getCostEstimat() {
		if (costEstimat == null) {
			this.costEstimat = this.estimeazaCost();
		}
		return costEstimat;
	}

	public Double getCostCalculat() {
		if (costCalculat == null) {
			this.costCalculat = this.calculCost();
		}

		return costCalculat;
	}

	// se allege metoda de estimare in functie de
	// valoarea atributului modelEstimare
	public Double estimeazaCost() {
		Double cost = 0.0;
		switch (modelEstimare)
		{
		case COCOMO:
			cost = this.modelCOCOMO();
			break;
		case ABC:
			cost = this.modelABC();
			break;
		case MonteCarlo:
			cost = this.modelMonteCarlo();
			break;
		}
		return cost;
	}

	// similar se implementeaza si alegerea metodei de calcul al costului

	// se implementeaza algoritmii specifici fiecarei metode de estimare
```

```
	public Double modelCOCOMO() {

		// implementarea algoritmului COCOMO
		return 1000.0;

	}

	public Double modelABC() {
		// implementarea algoritmului ABC
		return 1000.0;
	}

	public Double modelMonteCarlo() {

		// implementarea algoritmului Monte Carlo
		return 1000.0;
	}
}
```

ii. se creeaza cate o subclasa a clasei proiect pentru fiecare model/metoda de calcul si aplicarea polimorfismului, cele 2 metode estimeazaCost() si calculCost() avand implementari diferite in subclase. Clasa Proiect ar putea fi declarata abstracta.

Aceasta solutie nu mai este detaliata, fiind usor de intuit.

3. Dezavantajele solutiilor propuse.

Solutia i:

- ✓ aparitia unei noi metode de estimare/calcul ar impune modificarea clasei Proiect, in sensul adaugarii unei operatii (metode) noi si modificarea implementarii celor 2 operatii (metode) de alegere a metodelor de lucru – estimeazaCost() si calculCost(). Asa ceva nu este de dorit deoarece implica re-testarea si recompilarea aplicatiei;
- ✓ clasa Proiect ar avea o coeziune redusa (ar fi cam „grasuta”), ceea ce o face greu de testat, intretinut si impartit sarcinile pe programatori.

Solutia ii:

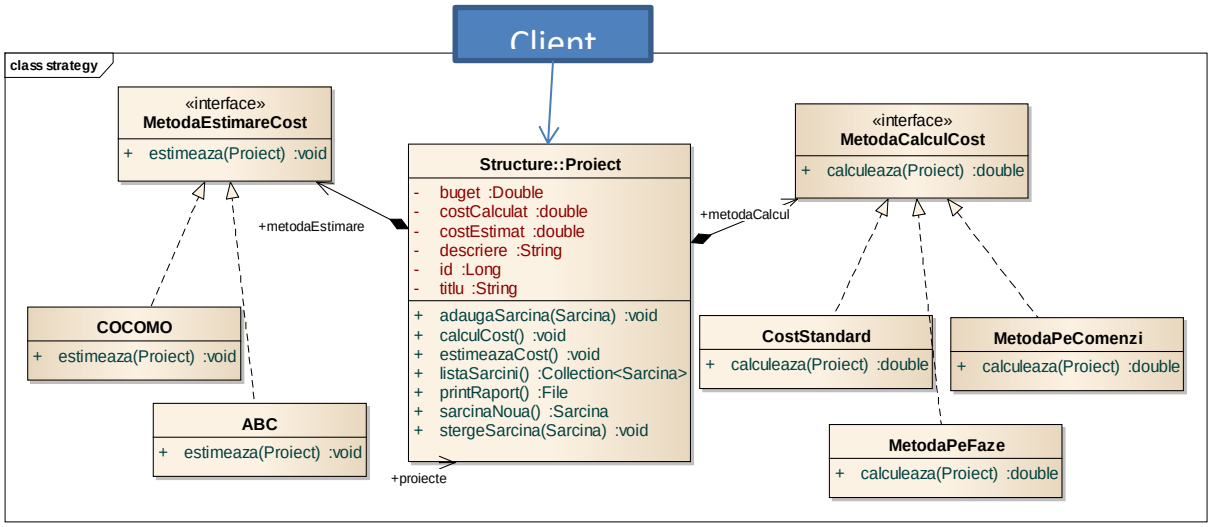
- ✓ schimbarea metodei de estimare sau calcul pentru un proiect este dificil de gestionat deoarece obiectul proiect va schimba subclasa;
- ✓ alegerea modelului/metodei de catre utilizator la run-time creeaza aceeasi problema a schimbarii subclasei de catre obiectul de tip proiect;
- ✓ in cazul problemei date, aceasta solutie este una complexa si dificil de implementat deoarece avem de-a face cu 2 ierarhii: una bazata pe metodele de estimare si una pe metodele de calcul al costului efectiv. Cum un proiect are asociata atat o metoda de estimare, cat si una de calcul, atunci va trebui sa se creeze cate o subclasa pentru fiecare combinatie posibila intre metodele de estimare si metodele de calcul (sub forma „metodaEstimare_metodaCalcul”. De exemplu, am avea subclasa COCOMO_DirectCosting, COCOMO_CostStandard, COCOMO_Faze, ABC_DirectCosting etc. Pentru 4 metode de estimare si 3 metode de clacul am putea avea 12 subclase, iar aparitia unei metode noi de calcul ar presupune crearea a 4 subclase noi!

4. **Solutia de aplicare a sablonului Strategy.**

Cea mai eleganta solutie presupune aplicarea sablonului de proiectare Strategy. In acest sens se parcurg urmatorii pasi:

- Se defineste cate o interfata pentru fiecare familie de algoritmi: una pentru metode de estimare – MetodaEstimareCost, si una pentru metode de calcul al costului efectiv – MetodaCalculCost. Cele 2 interfete declara metodele(operatiile) ce vor fi implementate de clasele concrete, estimeaza() si calculeaza(), care vor avea drept parametru un obiect de tip proiect, astfel incat sa existe referinta la el pentru a avea acces la datele necesare implementarii algoritmilor.
- Se creeaza cate o clasa (Concrete Strategy) pentru fiecare metoda de estimare, respectiv calcul cost efectiv, care vor implementa interfata corespunzatoare.
- Clasa Proiect va juca rolul de Context. Ea va utiliza cele 2 interfete pentru a invoca o metoda de estimare sau calcul cost. In acest sens, clasa proiect va avea 2 atribute de tipul celor 2 interfete, metodaEstimare si metodaCalcul.

Diagrama de clase rezultata prin aplicarea sablonului Strategy este prezentata mai jos.



Codul pentru aplicarea sablonului:

```
// Se definesc cele 2 interfete pentru cele 2 familii de algoritmi
public interface IMetodaEstimareCost {

	public Double estimeaza (Proiect proiect);
}

public interface IMetodaCalculCost {

	public Double calculeaza(Proiect proiect);
}

// Clasa Proiect va juca rolul de Context

// Se adauga cate un atribut de tipul celor 2 interfete pentru a
```

```java
// avea acces la implementarile algoritmilor din clasele ConcreteStrategy

public class Proiect {
	private Double buget;
	private IMetodaEstimareCost metodaEstimare;
	private Double costEstimat;
	private IMetodaCalculCost metodaCalcul;
	private Double costCalculat;
	private Long id;
	private String titlu;

	public void estimeazaCost() {
		this.costEstimat = this.metodaEstimare.estimeaza(this);
		System.out.println(this.costEstimat);
	}

	public void calculCost() {
		this.costCalculat = this.metodaCalcul.calculeaza(this);
		System.out.println(this.costCalculat);
	}
}


// Se ceeeaza clasa ENUM

public enum MetodaEstimare {
	COCOMO, ABC

}

// Se ceeeaza clasele ConcreteStrategy care vor implementa algoritmii

// specifici metodelor de estimare/calcul cost

public class Cocomo implements IMetodaEstimareCost {

	@Override
	public Double estimeaza(Proiect proiect) {
		// Se implementeaza algoritmul metodei COCOMO
		return 1000.0;
	}
}

public class ABC implements IMetodaEstimareCost {

	@Override
	public Double estimeaza(Proiect proiect) {
		// Se implementeaza metoda ABC
		return 2000.0;
	}
}

// Clasa cu rol de client care va invoca metoda corespunzatoare

// alegerii utilizatorului

public class ClientStrategy {

	public static void main(String[] args) {
```

```
            // clientul care poate fi o metoda intr-o clasa Controller
            MetodaEstimare met=MetodaEstimare.ABC;
            Proiect p=new Proiect();

            // Utilizatorul alege metoda de estimare, iar in functie de aceasta
            // alegere se va instantia clasa corespunzatoare
            switch (met)
            {
            case COCOMO:
                  p.setMetodaEstimare(new Cocomo());
                  break;
            case ABC:
                  p.setMetodaEstimare(new ABC());
                  break;
            // Similar se specifica si celelalte metode
            }
            p.estimeazaCost();
      }
}
```

5. **Comentarii privind solutia.**

Se poate sesiza usor ca dezavantajele asociate solutiilor initiale, identificate la punctul 3, au fost eliminate:

- ✓ Aparitia unei noi metode de estimare sau calcul ar presupune doar adaugarea unei clase ConcreteStrategy noi, fara a se face vreo modificare in clasa Proiect. In schimb, va trebui modificat clientul;
- ✓ Clasa Proiect este mai „supla” (coeziunea este mai buna) deoarece nu mai implementeaza cate o metoda pentru fiecare metoda de calcul sau estimare;
- ✓ Schimbarea metodei de estimare si/sau calcul pentru un proiect se poate face fara modificari in aplicatie;
- ✓ Alegerea metodei de estimare/calcul se poate face la run-time.

Alte comentarii:

Se poate lesne observa ca mai exista un inconvenient pentru solutia propusa: logica alegerii metodei de estimare/calcul in functie de optiunea utilizatorului la run-time este implementata in aplicatia client. Acest neajuns se manifesta prin modificarea clientului in cazul in care apare o noua metoda de estimare/calcul, dar si prin eventuala redundanta a codului daca vor exista mai multi clienti care sa invoce aceste metode.

Pentru eliminarea acestui neajuns se recomanda aplicarea sablonului Abstract Factory.

Nota! Aceasta problema ar putea constitui al doilea exemplu de aplicare a unui sablon de proiectare (creational in acest caz). Se incurajeaza formularea de probleme care sa presupuna aplicarea a 2 sabloane!!!