Dizajn šabloni

Autori:
Milan Segedinac
Goran Savić

1. Dizajn šabloni

Dizajn softvera je proces u kome specificiramo globalnu strukturu softverske aplikacije koju razvijamo. Prilikom razvoja softvera često se susrećemo sa problemima i situacijama koji *liče* na neke koje smo već rešavali. U tim situacijama ne bismo mogli ponovo da iskoristimo softver koji smo već razvili, ali nam je već poznata globalna struktura softvera koji razvijamo. To znači da, iako ne možemo direktno da preuzmemo programski kod, možemo da preuzmemo rešenja koja smo primenili prilikom dizajna softvera.

Pojam dizajn šablona preuzet je iz arhitekture u kojoj ga je Kristofer Aleksander definisao na sledeći način: „Svaki šablon opisuje problem koji se iznova i iznova pojavljuje u našem okruženju i opisuje jezgro rešenja tog problema na takav način da rešenje možemo koristiti milion puta, a da nikada ne uradimo istu stvar dva puta.“ U objektno orjentisanom programiranju dizan šablon je *opis interakcija klasa i objekata namenjen za rešavanje generalnog problema koji se može prilagoditi konkretnim situacijama*. Vidimo da dizajn šablon nije gotov softverski dizajn koji može da se direktno prevede u programski kod, nego je templejt koji može da se iskoristi u dizajnu softvera.

Tipično je da se dizajn šabloni dele u tri grupe

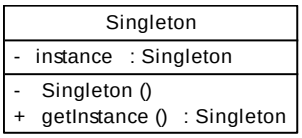
1. Šabloni kreiranja – definišu način kreiranja objekata. U ovu grupu spadaju *Abstract factory pattern*, *Builder pattern*, *Factory method pattern*, *Prototype pattern* i *Singleton pattern*
2. Šabloni strukture – definišu veze između objekata. U ovu grupu spadaju *Adapter pattern*, *Aggregate pattern*, *Bridge pattern*, *Composite pattern*, *Decorator pattern*, *Extensibility pattern*, *Facade pattern*, *Flyweight pattern*, *Marker pattern*, *Pipes and filters*, *Opaque pointer* i *Proxy pattern*.
3. Šabloni ponašanja – definišu tipične načine komunikacije između objekata. U ovu grupu šablona spadaju *Chain of responsibility pattern*, *Command pattern*, *"Externalize the Stack" pattern*, *Interpreter pattern*, *Iterator pattern*, *Mediator pattern*, *Memento pattern*, *Null Object pattern*, *Observer pattern*, *Weak reference pattern*, *Protocol stack*, *Scheduled-task pattern*, *Single-serving visitor pattern*, *Specification pattern*, *State pattern*, *Strategy pattern*, *Template method pattern* i *Visitor pattern*.

Detaljan pregled svih navedenih dizajn šablona značajno prevazilazi opseg ovog materijala. Čitaoca koji je zainteresovan za detaljniji pregled dizajn šablona upućujemo na knjigu *Design Patterns: Elements of Reusable Object-Oriented Software* čiji su autori Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides i Grady Booch. U ovom poglavlju biće dat pregled dva dizajn šablona – singleton i command – i njihova upotreba će biti ilustrovana na primeru jednostavne Java aplikacije – kalkulatora.

Singleton šablon

Singleton šablon je šablon kreiranja inspirisan matematičkim konceptom singleton skupa. Singleton skup je *skup koji ima tačno jedan element*. U objektno-orjentisanom programiranju *singleton je klasa koja ima tačno jednu instancu*. Česte su situacije u kojima je ovo slučaj: studentska služba treba da ima tačno jedan spisak studenata, aplikacija treba da ima tačno jednu konfiguraciju, aplikacije koje dele štampač trebaju da pristupaju tačno jednom programu za raspoređivanje štampanja (printer spooler). Iz navedenih primera možemo da uočimo da se singleton šablon koristi kada imamo *deljeni resurs* kome trebaju da pristupe različiti delovi programa ili različiti programi.

Struktura singleton šablona prikazana je na slici ispod.



Slika – singleton šablon

Vidimo da je konstruktor Singleton klase privatni, što znači da će moći da se pozove jedino u okviru ove klase. Umesto konstruktora, kada nam zatreba instanca Singleton klase pozivamo statičku javnu metodu getInstance. Implementacija Singleton klase u programskom jeziku Java data je listingom ispod.

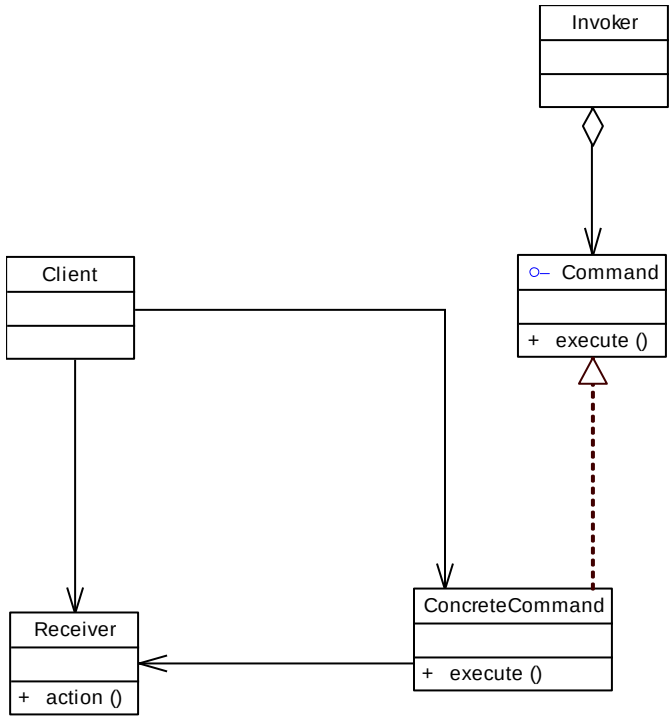
```
public class Singleton {
	private Singleton(){ }

	private static Singleton instance = new Singleton();

	public static Singleton getInstance(){
		return instance;
	}

}
```

Slika – singleton klasa

Kao što vidimo u programskom kodu u listingu iznad, postojanje tačno jedne instance je ostvareno pomoću atribut `instance`. Prilikom učitavanja klase Singleton, atributu instance se dodeli vrednost. Obratite pažnju da pri tome konstruktor može da se izvrši iako je privatni, jer je pozvan u okviru klase. Pri svakom pozivu statičke metode `getInstance` biće vraćen jedini objekat klase Singleton – onaj na koji referencu čuva atribut `instance`.

Command šablon

Command šablon je šablon ponašanja u kome se objekat koristi za enkapsulaciju svih informacija potrebnih za izvršavanje neke akcije. Te informacije uključuju metodu koja će se izvršiti, objekat koji poseduje metodu koja se izvršava i listu argumenata potrebnih za izvršavanje ove metode. Struktura command šablona data je na dijagramu ispod.



Slika – Command šablon

Interfejs `Command` je zajednički za sve komande i ima metodu `execute` koja će se pozvati prilikom izvršavanje zahtevane komande. Klasa `Receiver` čuva informacija potrebne za izvršavanje komande i ima metodu `action` koja će se obaviti prilikom izvršavanje komande. Klasa `Invoker` ima listu komandi koje može da izvrši pozivom `execute` metoda. Objekat klase `Client` kreira objekte klase `ConcreteCommand` i za njih postavlja objekte klase `Receiver`. Klasa `ConcreteCommand` definiše vezu između akcije koja će se izvršiti i objekta klase `Receiver` nad kojim će se ta akcija izvršiti.

Korišćenje ova dva šablona ilustrovaćemo na primeru kalkulatora. Interfejs komande prikazan je listingom ispod.

```
public interface Command {
    public void execute();
    public void unExecute();
}
```

Listing – `Command` interfejs

Pored metode `execute`, koju predviđa command šablon, dodali smo još jednu metodu: `unExecute`. Ova metoda će nam omogućiti da jednostavno implementiramo undo funkcionalnost (poništavanje prethodne operacije).

Receiver u ovoj aplikaciji je klasa `Calculator` prikazana listingom ispod.

```
public class Calculator {

	private double current = 0;

	public void operation(char operator, double operand) {
		if (operator == '+') {
			current += operand;
		} else if (operator == '-') {
			current -= operand;
		} else if (operator == '*') {
			current *= operand;
		} else if (operator == '/') {
			current /= operand;
		}
		System.out.println("Current value = " + current + " after " +
				operator + " " + operand);
	}
}
```

Listing – Receiver klasa `Calculator`

Klasa Calculator čuva trenutnu vrednost izračunavanja (atribut `current`) i modeluje operacije sabiranja, oduzimanja, množenja i deljenja metodom `operation`. Nakon svake izvršene operacije biće prikazana trenutna vrednost atributa `current` i operator i operand. Ovu klasu kao receiver koristi klasa koja modeluje konkretnu komandu: `CalculatorCommand`. Ta klasa je prikazana listingom ispod.

```
public class CalculatorCommand implements Command {

	private char operator;

	private double operand;

	private Calculator calculator;


	@Override
	public void execute() {
		calculator.operation(operator, operand);
	}

	@Override
	public void unExecute() {
		//ponistavanje komande je izvrsavanje njene inverzne operacije
		calculator.operation(inverse(operator), operand);
	}


	private char inverse(char operand){
		if(operand == '+'){
			return '-';
		}
		else if(operand == '-'){
			return '+';
		}
		else if(operand == '*'){
			return '/';
		}
		else if(operand == '/'){
			return '*';
		}
		else{
			return ' ';
		}
	}
}
```

Listing – Konkretna komanda modelovana klasom `CalculatorCommand`.

Klasa `CalculatorCommand` čuva operator i operand i ima referencu na instancu klase `Calculator` kojim treba da se izvrši komanda. Ova klasa implementira interfejs `Command`, što znači da ima metoda `execute` i `unExecute`. Metoda `execute` se svodi na poziv operacije nad kalkulatorom za zadati operator i operand. Metoda `unExecute` treba da obezbedi poništavanje komande. U aritmetičkim operacijama poništavanje je izvšravanje inverznog operatora (- za +, + za -, * za / i / za *) nad rezultatom za isti operand. Stoga je bilo potrebno implementirati metodu `inverse`, koja vraća inverzni operator. Uz implementaciju `inverse` metode, poništavanje komande se svodi na izvršavanje te komande za inverzni operator.

Klasa koja ima ulogu invoker-a i client-a je `Application`. Ova klasa čuva listu konkretnih komandi, ali i ima objekat klase `Calculator` (receiver). Da bismo obezbedili da naš program ima sigurno tačno jednu aplikaciju, implementirali smo je kao singleton šablon. Ta klasa je prikazana listingom ispod.

```
public class Application {

	private Application() {
		calculator = new Calculator();
		commands = new ArrayList<Command>();
	}

	private static Application instance = new Application();

	public static Application getInstance() {
		return instance;
	}

	private Calculator calculator;
	private ArrayList<Command> commands;

	private int current = 0;

	public void compute(char operator, double operand) {
		// kreiranje i izvrsavanje komande
		Command command = new CalculatorCommand(calculator, operator,
							operand);
		command.execute();

		// dodavanje komande u listu, da bi kasnije mogla da se ponisti
		commands.add(command);
		current++;
	}

	public void undo(int levels) {
		System.out.println("\n---- Undo " + levels + " levels ");
		// ponistavanje komandi
		for (int i = 0; i < levels; i++) {
			if (current >= 0) {
				// preuzimanje komande iz liste
				Command command = commands.get(--current);
				// ponistavanj komande
				command.unExecute();
			}
		}
	}

	public void redo(int levels) {
		System.out.println("\n---- Redo " + levels + " levels ");
		for (int i = 0; i < levels; i++) {
			if (current < commands.size()) {
				Command command = commands.get(current++);
				command.execute();
			}
		}
	}
}
```

Listing – Klasa Application

Poništavanje i ponovno izvršavanje komandi u aplikaciji realizovano je pomoću liste komandi. U ovoj listi čuvaju se sve komande koje se izvršavaju u toku rada programa. Stoga izvršavanje komande (metoda `compute`) kreira novu komandu, izvrši je i doda je u listu.

Atribut `current` čuva trenutnu poziciju u listi komandi. Poništavanje komandi realizuje se pomoću metode `undo`. Ova metoda primi broj komandi koje treba poništiti, prođe od kraja ka početku liste komandi (onoliko koraka koliko komandi trebamo da poništimo) i za jednu po jednu komandu poziva njenu metodu `unExecute`. Pri tome se ažurira atribut `current`.

Pored toga što možemo da poništimo komande, poništene komande možemo i ponovo da izvršimo. To je modelovano metodom `redo`. Ova metoda polazi od `current` pozicije u listi komandi, i nad jednom po jednom komandom izvršava njenju `execute` metodu.