Refaktoring koda

Refaktoring (eng. refactoring) je Martin Fowler definirao kao “promjene interne strukture softvera da bi bio lakši za razumijevanje i jeftiniji za modificiranje bez promjene njegovog ponašanja” (Fowler 1999). Riječ “refactoring” u modernom programiranju nastala je od originalne riječi “factoring” koju je uveo inženjer Larry Constantine u strukturiranom programiranju, a koja referencira na dekompoziciju programa u konsititucione dijelove onoliko koliko je to moguće (Yourdon I Constantine 1979).

Refaktoring se može definirati i kao tehnika restrukturiranja koda na disciplinirani način. To je iterativni način poboljšanja koda.

1. A class doesn’t do very much (Klasa ne radi puno) – Kada klasa ne radi mnogo toga potrebno je provjeriti da li su sve odgovornosti klase dodijljene drugim klasama i eliminirati klasu u potpunosti. Klasa Lokacija ne radi ništa tako da ćemo je eliminirati pri čemu moramo izmijeniti kod tamo gdje se ta klasa koristila. Umjesto instanci klase uvesti ćemo atribut lokacija koji će biti tipa Tuple<double, double>.

2. Code is duplicated (Kod je dupliciran) – Dupliciran kod predstavlja većinom prvi faktor greške u dizajnu jer zahtjeva paralelnu modifikaciju – kada se urade promjene na jednom mjestu, moramo raditi promjene i na drugom mjestu. U klasama UpdateProfile i AdminUpdate vrši se validacija korisničkog imena i passworda pomoću metoda iz odgovarajućih ViewModela pri čemu se kod duplicira jer je validacija ista te da bi izbjgli dupliranje koda uveli smo novu klasu ValidacijaUser sa svim potrebnim metodama za validaciju.

3. Rename a variable with a clearer or more informative name (Preimenovati varijablu sa više informativnim imenom) - Kada ime varijable nije jasno potrebno ga je promijeniti. Isti zahtjev se primjenjuje na konstante, klase i rutine. Prošli smo još jednom kroz kod i promijenili imena varijabli gdje je to bilo potrebno tako da sve varijable imaju informativno ime (npr. textBox3 preimenovan u emailTextBox i sl.). Promjene su odrađene u klasi UpdateProfil za svaki textBlock. Ovo nam može olakšati prevođenje labela u budućnosti, jer npr. usernameTextBlock nosi više informacija od textBlock3 i sl.

4. Global variables are used (Globalne varijable se koriste)

5. A parameter list has too many parameters

6. Replace a magic number with a named constant (Zamijeniti ‘magične’ brojeve sa imenovanom konstantom)

Design paterni

1. **Adapter patern - Strukturalni patern**

   Adapter patern možemo koristiti da omogućimo širu upotrebu već postojećih klasa kada nam je potreban drugačiji interfejs već postojeće klase, a ne želimo mijenjati postojeću klasu. Novokreirana adapter klasa služi kao posrednik između originalne klase i željenog interfejsa.

   Potrebno je napraviti novi interfejs Itarget (zahtjevani interfejs) kojeg implementira klasa Adapter koja prilagođava stari interfejs. Klasa Adaptee definira već postojeći interfejs kojeg prilagođavamo. U klasi ITarget definišemo metode koje treba izmijeniti. Klasa adapter implementira te metode na odgovarajući način s ciljem da se postigne zahtjevani interfejs.

   Ovaj patern bi se mogao iskoristiti kod prikaza statistike. AdminStatistikaViewModel definiše metode koje vraćaju statistiku o korisnicima (procenat registrovanih i neregistrovanih korisnika) u vidu decimalnog broja sa preciznosti od dva decimalna mjesta. Ako bi htjeli promijeniti ispis željenih podataka možemo definisati novi interfejes IAdminStatistka sa metodama newGetPrecentageOfRegistered() i newGetPrecentageOfUnregistered() koje ćemo implementirati u Adapteru na način koji nama odgovara. Učinjene su potrebne izmjene tako da je u adapteru omogućen ispis statistike na jednu decimalu.

2. **Iterator patern – Patern ponašanja**

   AOvaj pattern se koristi kada je potrebno imati uniforan način pristupa bilo kojoj kolekciji. Ako recimo želimo iz nekog razloga da primimo ArrayList, Array i HashMapu, možemo iskoristiti iterator interface pomoću kojeg ćemo najbolje omogućiti uniforan pristup, skratiti kod, napraviti bolji polimorfizam.

   Potrebno je napraviti interface Iterator (ne mora nužno taj naziv) i naslijediti sve naše klase iz tog interface-a, na kraju taj interface ćemo realizirati u nekoj drugoj klasi pomoću koje ćemo realizovati specifične metode tog interface-a. Naravno svaka klasa koja naslijedi Interface metodu će vraćati Interface. (ref. Java pristup)

   Ukoliko imamo rad sa više vrsta kolekcija implementacija ovog paterna je veoma korisna. Pošto u našoj implementaciji imamo samo listu upotreba ovog paterna nije od nekog značaja. Mogao bi se iskoristiti u nekoj posebnoj klasi koja bi imala realizovan Iterator interface i imala npr. metodu koja će raditi nešto sa elementima svih kolekcija. Patern je iskorišten u klasi RestoraniLista.