Antipaterni u razvoju softvera

Uroš Milenković 1130/2015

Refaktorisanje softvera

Cilj razvoja antipaterna je da se opišu postupci softverskog refaktorisanja

Softversko refaktorisanje predstavlja modifikaciju koda radi poboljšanja strukture radi lakšeg održavanja

Formalno refaktorisanje se sastoji od:

1. Apstrakcija super klase
2. Eleminisanje uslova
3. Agregatna apstrakcija

Antipaterni u razvoju softvera

Blob

Lava

Dvosmislen pogled

Zlatni čekić 🔨

Dugi manji antipaterni

Blob

Poznat kao i: Bog svih klasa

Najčešće skaliranje: Aplikativni nivo

Ime refaktorisanog rešenja: Refaktorisanje odgovornosti

Tip refaktorisanja: Softver

Uzrok: Lenjost

Dokaz: “Ova klasa je sama *srž* naše aplikacije”

Dobio ime po starom horor filmu “The Blob”, čudovište koje redom “guta” sve na šta naiđe

Kao takav može da preuzme kontrolu nad celom aplikacijom pa i da proguta celu objektno-orijentisanu arhitekturu

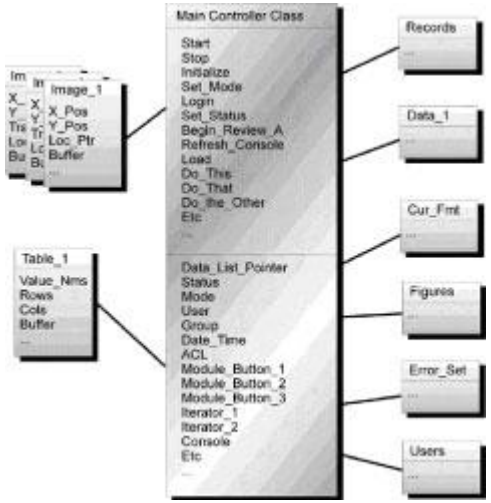
Blob (2)

Nalazi se u dizajnu gde jedna klasa ima monopol nad celim procesom a ostale klase enkapsuliraju podatke

U suštini, Blob predstavlja proceduralni dizajn iako može da se sakrije iza OO arhitekture

Često se javlja u situacijama gde neki dokaz koncepta prerasta u produkcioni sistem

Jezici koji su orijentisani prema korisničkom interfejsu često dozvoljavaju implementaciju logike u okviru forme pa je to čest slučaj gde Blob nastaje



“Blob” kontroler sa “pomoćnim” klasama za enkapsulaciju podataka

Blob – simptomi

Jedna klasa sa velikim brojem atributa i metoda

Jedan kontroler sa velikim brojem manjih klasa za reprezentaciju podataka

Nedostatak OO dizajna - “main loop” se nalazi u Blob klasi

Blob ne omogućava da se nasledi

Modifikacija iziskuje veliku promenu u ostalim klasama

Promena u ostalim klasama zahteva promenu u Blobu

Nemoguće je iskoristiti ga ponovo i testirati

Kao takav, učitan u memoriju može zauzeti velike resurse čak i za male operacije

Uzroci

Nedostatak OO dizajna - dizajneri nemaju dobro poznavanje u oblasti OO

Nedostatak bilokakve arhitekture - nedostatak definicije sistema

Zakasnela intervencija - neadekvatan pregled napretka

Ograničena intervencija - razvijaocima je nekada uskraćeno da prave nove klase već dodaju male funkcionalnosti

Refaktorisano rešenje

Kluč je ukloniti Blobovo ponašanje

Prebaciti odgovornost na druge klase tako da one dobiju veći smisao

1. Prepoznati skup operacija i atributa koje predstavljaju konflikt
2. Razvrstati konfliktne kolekcije u odvojene klase
3. Ukloniti redudantne veze ka kontroleru
4. Uklanjamo sve tranzitivne veze

Lava

Poznat kao i: Mrtav kod

Najčešće skaliranje: Aplikativni nivo

Ime refaktorisanog rešenja: Konfiguracioni menadžment

Tip refaktorisanja: Proces

Uzrok: Pohlepa, lenjost

Dokaz: “Ova klasa je sama *srž* naše aplikacije”

Dokaz: “Pa taj deo koda su pisali Pera i Žika, koji više nisu u firmi, dok je Miloš (koji je otišao iz firme pre dva meseca) pokušavao da ga zaobiđe preko nečeg drugog za šta nisam siguran. Niko od njih to nije dobro dokumentovao i nisam siguran da li taj kod uopšte i radi”

Lava (2)

30% - 50% koda velikih sistema koji pravi probleme su loše dokumentovani ili se ne razumeju

Ovakav antipatern se često nalazi u projektima koji su započeli kao istraživanje a završili u produkciji

Takav kod razvijaoci uglavnom ostavljaju da “sakuplja prašinu” jer su možda veoma važni a ne prave štetu

“Architecture is the art of how to waste space.”

—Phillip Johnson

Lava (3)

Lava kod se teško analizira, verifikuje i testira, što je u praksi često i nemoguće

Lava kod može biti skup za učitavanje u memoriju i da utiče na performanse

Gubi se mogućnost modularizacije i ponovne iskorišćenosti koda

Simptomi i posledice

Nedokumentovane kompleksne funkcije, klase ili segmenti koda koji izgledaju bitno

Blokovi iskomentarisanog koda

Kod koji se ne koristi a ostavljen je da “lebdi” sa strane

Nekorišćeni interfejsi u hederu

Ako kod ostane može se profilisati kao dobar kod i iskoristiti negde drugde

Ako se Lava ne primeti na vreme može eksponencijalno da raste

Uzroci

Jedan programer je pisao kod

Nedostatak arhitekture

Nekontrolisana distribucija neproverenog koda

Kada se došlo do zaključka da polazna arhitektura nije dobra pa je došlo do rekonfiguracije

Repetitivni proces

Refaktorisano rešenje

Jedino pravo rešenje je da arhitektura prethodi produkciji

Ako je arhitektura osmišljena za kratak rok onda ona iza sebe ostavlja dosta koda koji se ne koristi

Proučavanje međuzavisnosti koda

Uspostaviti interfejse na sistemskom nivou

Alati za praćenje verzija pomažu u izbegavanju Lava koda

Dvosmislena pogled

OO analiza i dizajn su često prisutni bez definisanja pogleda s kojeg se posmatra model

Pomešani pogledi na model ne omogućavaju fundamentalno razdvajanje interfejsa od implementacionih detalja

Postoje tri pogleda na model

1. Poslovni
2. Specifikacioni
3. Implementacioni

Modeli mogu više smetati nego imati korist ako se ne fokusiraju na traženu perspektivu

Zlatni čekić

Poznat kao i: Glava u pesku

Najčešće skaliranje: Aplikativni nivo

Ime refaktorisanog rešenja: Refaktorisanje odgovornosti

Tip refaktorisanja: Softver

Uzrok: Neznanje, ponos, zatucanos

Dokaz: “Naša baza je naša arhitektura.”

Jedan od najčešćih antipaterna u industriji

Proizvođač nekog rešenja će uglavnom zastupati stava da će njegovo rešenje rešiti sve probleme

Kada tim programera dobije veliku nadležnost za neki produkt onda vide svoj alat kao nešto čime je najbolje da se problem reši

Simptomi

Koriste se isti alati i proizvodi za širok spektar različitih proizvoda

Rešenje ima bolje performanse od ostalih na tržištu

Arhitektura se najbolje opisuje kao set alata

Novi razvoj zavisi od određenog proizvođača ili tehnologij

Programeri su izolovani od industrije

Uzroci

Nekoliko uspešnih projekata je koristilo te alate

Veliko ulaganje u treniranje za korišćenje alata

Grupa izolovana od industrije, drugih kompanija

Oslanjanje na vlasničke proizvode koji nisu dostupni u drugim proizvodima u industriji

Rešenje

Zahteva promenu filozofije i procesa razvoja

Programeri moraju da budu posvećeni istraživanju novih tehnologija

Programeri moraju da budu u toku sa trendovima u tehnologiji, uopšte

Posećivanje konferencija

Menadžment treba da usvoji otvorene sisteme i arhitekture

Fleksibilan, ponovo iskorišćen kod zahteva analizu inicijalnog razvoja

Zapošljavanje ljudi iz različitih oblasti i sa različitom pozadinom

Menadžment mora da nagradi programere koji samoinicijativno poboljšavaju svoj posao

Sidro ⚓

Sidro je deo softvera ili hardvera koji nema upotrebnu vrednost na trenutni projekat

Posećenost proizvodu je napravljena bez prethodnog tehničkog izučavanja

Posledica takvog ponašanja je da se previše vremena posvećuje nečemu što nije potrebno

Dobra praksa je da postoji tehnička alternativa svakoj tehnologiji kako bi se sagledala šira slika

Korišćenjem privremenih licenci za softver se otklanjaju pocetničke greške

Pravljenje prototipova preko ovih licenci najbolje prikazuje da li nam je takva vrsta tehnologije potrebna

Copy – Paste programiranje

Poznat kao i: Kloniranje softvera

Najčešće skaliranje: Aplikativni nivo

Tip refaktorisanja: Softver

Uzrok: Lenjost

Dokaz: “Mislio sam da sam popravio ovaj bag.”, “Napravili smo 400,000 linija koda za mesec i po dana”

Česta je praksa kopirati kod, koja vodi do noćnih mora za održavanje

Proizilazi iz pretpostavke da je lakše modifikovati postojeći kod nego napisati ga ispočetka

Simptomi i posledice

Isti bag se propagira na više mesta, gde god je kopiran kod

Broj linija koda se povećava bez imalo uloženog truda

Teško se identifikuju i otklanjaju problemi na više različitih mesta

Kod se ponovo upotrebljava bez imalo napora

Ovaj antipatern vodi do skupog održavanja

Kod koji se koristi na više mesta nije konvertovan u dokumentovanu i laku formu

Razvijaoci prave više metoda za popravljanje bagova umesto jedne koja rešava sve probleme

Uzroci

Pravljenje koda koji je moguce ponovo koristiti zahteva vremensko ulaganje, dok investitori uglavnom razmišljaju na kratkoročno

Kontekst koji se krije iza koda nije sačuvan

Nedostatak apstrakcije i razumevanje nasledjivanja, kompozicije i drugih strategija

Komponente nisu dobro dokumentovane i čitljive za programera

Kada ljudima nisu poznati novi alati i tehnologije

Rešenje

Ako je projekat nov postoje 2 strategije za izbegavanje kopiranja koda

1. Napraviti klase koje se mogu lako naslediti
2. Organizovati kod u komponente

Kod prvog metoda je problem što korisnik klase mora biti bar malo upoznat u implementaciju da bi je nasledio

Kod drugog objekti se koriste takvi kakvi su i pristupa im se preko interfejsa

Ako u projektu već postoji ovaj antipatern treba primenjivati sledeće korake

1. Detektovati gde se javlja
2. Refaktorisati
3. Organizovati kod tako da kasnije može ponovo da se iskoristi