Zbornik radova Fakulteta tehničkih nauka, Novi Sad

**UDK: 7.05:62**
**DOI: https://doi.org/10.24867/11BE01Grujic**

# AUTOMATSKA DETEKCIJA INDIKATORA LOŠE DIZAJNIRANOG KODA BAZIRANA NA INFORMACIJAMA EKSTRAHOVANIM IZ TEKSTUALNOG SADRŽAJA

# AUTOMATIC CODE SMELL DETECTION BASED ON INFORMATION EXTRACTED FROM THE TEXTUAL CONTENT OF THE CODE

Katarina-Glorija Grujić, *Fakultet tehničkih nauka, Novi Sad*

**Oblast – ELEKTROTEHNIKA I RAČUNARSTVO**

**Kratak sadržaj –** *Radovi koji se bave automatskom detekcijom loše dizajniranog koda (eng. Code smell) već postoje. Međutim, ti radovi uglavnom uzimaju u obzir samo par tipova loše dizajniranog koda i njihova detekcija je jako zavisi od programskog jezika u okviru kog se detektuju. Pored toga, većina radova se oslanja na strukturalne metrike i samim tim potrebno je definisati razne pragove (eng. threshold) kako bi se detektovao indikator loše dizajniranog koda. Samim tim, rezultati mogu varirati u odnosu na projekte nad kojima se detektuje iz razloga što za svaki projekat se mora posebno definisati prag tih metrika. U ovom radu, loše dizajniran kod se detektuje isključivo na onovu informacija dobijenih iz tekstualnog sadržaja - kod. Ukoliko se koristi sam kod aplikacije, nestaje potreba definisanja pragova za razne metrike, jer se obrada vrši nad prirodnim jezikom za svaki projekat posebno.*

**Ključne reči:** *Obrada prirodnog jezika, embedding algoritmi, bert, word2vec, loše dizajniran kod*

**Abstract** – *Researches in the field of the automatic detection of code smells already exist. However, these researches generally take into account only a few types of code smells, and their detection is highly dependent on the programming language within which they are detected. In addition, most papers rely on structural metrics, and therefore it is necessary to define various thresholds in order to detect the code smells. Therefore, the results may vary on the projects over which it is detected for the reason that the threshold of these metrics must be defined separately for each project. In this paper, code smells is detected exclusively by information obtained from textual content - code. If used alone in the application, there is no need to define thresholds for various metrics, because the processing is done over natural language for each project separately.*

**Keywords:** *Natural language process, embedding, bert, word2vec, code smells*

## 1. UVOD

Softver bogat funkcionalnostima je očigledan cilj programera, jer na taj način softver dobija na značaju zainteresovanoj strani.

**NAPOMENA:**
**Ovaj rad proistekao je iz master rada čiji mentor je bio prof. dr Aleksandar Kovačević.**

Međutim, postoji važniji, manje vidljiv posao koji programer mora redovno da obavlja, a to je održavanje kvaliteta programskog koda [1]. Kod niskog kvaliteta je krh i krut, što znači da je podložan greškama, lako se kvari i teško ga je promeniti tokom vremena [2]. Iz tog razloga kasniji razvoj postaje skuplji, a u nekim slučajevima čak i nemoguć. S obzirom na to da održavanje i razvoj softvera može predstavljati i do 80% troškova [3][4][5], povećanje kvaliteta koda bi značilo da se značajno smanje opšti troškovi razvoja softvera.

Kako bi stvorili pouzdan i održiv softver, programeri moraju redovno vršiti refaktorisanje koda. Refaktorisanje podrazumeva restrukturiranje koda radi uklanjanja indikatora loše dizajniranog koda, bez promene ponašanja softvera. Osnovna aktivnost u razvoju softvera je osiguravanje dugotrajnosti softverskog rešenja [6]. Iako refaktorisanje zahteva napor i ne predstavlja razvoj funkcionalnosti, troškovi koji nastaju zanemarivanjem zdravlja koda su znatno veći od cene kontinualnog refaktorisanja [1][6].

Većina predloženih tehnika za detekciju indikatora loše dizajniranog koda zasniva se na heuristikama. Izračuna se skup metričkih vrednosti izvornog koda, a zatim se na osnovu unapred definisanih pragova (eng. *threshold*) odredi da li nešto jeste *code smell* ili ne [7]. Ovakvi pristupi imaju nekoliko ograničenja koje sprečavaju njihovo korišćenje u praksi [7][8]. Najveći problem predstavlja neusaglašenost postojećih detektora i zavisnost od parametara definisanih od strane korisnika, za koje ne postoji precizan način definisanja.

Iz tog razloga se počelo ekperimentisati koristeći tehnike zasnovane na mašinskom učenju (eng. *machine learning - ML*) za detekciju *code smell*-ova. Uglavnom se koriste tehnike učenja pod nadzorom (eng. *supervised learning*). To znači da *ML* model uči mapiranje skupa prediktora (nezavisne promenljive) na skup zavnisnih promenljivih. Zavisne promenljive mogu biti binarne - postoji indikator loše dizajniranog koda ili ne, a mogu biti i izražene u jačini prisustva indikatora - pomoću unapred definisane skale. Ovakav pristup učenja obećava da će rešiti ograničenja tehnika zasnovanim na heuristikama [8].

U ovom radu detekcija je vršena koristeći isključivo prirodni jezik sadržan u imenima identifikatora i komentara. Ulaz u sistem predstavlja deo koda za koji se vrši detekcija, a izlaz označava da li je taj deo koda afektovan (izlaz je binaran). Rad predstavlja nadogradnju rada [9] u kom je predstavljena implementacija *TACO* alata koji takođe koristi prirodni jezik za detekciju 5 različitih indikatora loše dizajniranog koda.

Evaluacija je vršena nad skupom podataka koji je predstavljen u radu [10]. Pored poređenja performansi algoritma sa radom [9] čiju nadogradnju predstavlja ovaj rad, performanse su poređenje i sa radom [11], čiji pristup trenutno pruža najbolje rezultate u polju detektovanja indikatora loše dizajniranog koda [10].

Rad je organizovan na sledeći način: Poglavlje 2 opisuje radove koji su se bavili problemom automatskog detektovanja indikatora loše dizajniranog koda, njihove algoritme, rezultate i predstavlja sličnosti opisanih radova sa ovim. Treće poglavlje govori o skupu podataka nad kojim je vršena evaluacija, kao I o specifikaciji i implementaciji sistema. Četvrto poglavlje govori u rezultatima eksperimenta i opisuje način vršenja evaluacije. Peto poglavlje predstavlja zaključak, kao i ideje za dalje unapređenje sistema.

## 2. PRETHODNA REŠENJA

U radu [9] indikatori loše dizajniranog koda se detektuju koristeći prirodan jezik, što u slučaju softvera predstavlja sam kod. Pretprocesiranje se vrši koristeći proces normalizacije teksta koji je uobičajen za oblast pretraživanja informacija (eng. *information retrieval normalization process*). Za dobijanje informacija korišćeni su komentari i identifikatori promenljivih u kodu. Nakon što su izvučeni tokeni iz teksta, primenjuje se heuristika bazirana na domenskom znanju u zavisnosti od toga koji *code smell* se detektuje. Sve heuristike zasnovane su na meri sličnosti između delova koda (metoda, klasa, deo metode, deo klase, itd.). Koristeći *Latent Semantic Indexing* (*LSI*) algoritam deo koda se modeluje kao vektor, na osnovu broja ponavljanja termina u posmatranom delu koda. Na kraju se računa kosinusna sličnost između dobijenih vektora iz *LSI* algoritma. Za evaluaciju koristili su preciznost, odziv i F-meru. Rezultati koji su dobili koristeći ovaj alat pokazuju da preciznost varira između 63% i 77%, odziv varira između 71% i 96%, dok F-mera varira između 67% i 82% u zavisnosti od projekta na kojem je vršeno testiranje.

Obzirom na napredak oblasti procesiranja prirodnog jezika (eng. *natural language proccessing*) i reprezentacije teksta (eng. *text embedding*), pristup koji je korišćen u radu [9] moguće je nadograditi koristeći algoritme za reprezentaciju teksta koji trenutno daju najbolju reprezentaciju i koriste se u oblasti obrade prirodnog jezika. Iz tog raloga, korišćeni su algoritmi *word2vec* i *BERT*.

U radu [11] je detektovano 4 antipaterna[1]: *blob, functional decomposition, spaghetti code* i *swiss army knife*, kao i 15 indikatora loše dizajniranog koda koji čine njihovu osnovu. Prepoznavanje je vršeno korišćenjem strukturalnih metrika. Alat koristi pristup zasnovan na unapred definisanim pravilima. Skup podataka korišćen za evaluaciju alata predstavlja 10 aplikacija otvorenog koda iz različitih domena. Za meru evaluacije koristili su preciznost i odziv. Rezultate koji su dobili koristeći ovaj alat pokazuju da njegova preciznost varira između 36.7% i 83.4% u zavisnosti od projekta na kojem je vršeno testiranje. Zbog sporog izvršavanja algoritma, odziv je prikazan samo za jedan projekat i iznosi 100%.

[1] Antipatern opisuje loš ponavljajući dizajn koji negativno utiče na kvalitet softvera [13], dok indikator loše dizajniranog koda ukazuje na moguće postojanje antipaterna.

## 3. METOD

U narednim poglavljima izloženi su skup podataka i arhitektura sistema.

### 3.1. Skup podataka

Skup podataka korišćen u ovom radu je skup predstavljen u radu [10]. Sadrži 30 *Java* projekata otvorenog koda koji su manuelno labelirani za 13 različitih tipa indikatora loše dizajnirang koda. Iako nisu u celosti manuelno označavani, autori su koristili proceduru označavanja koja smanjuje mogućnost pojave delova koda koji predstavjaju indikator loše dizajniranog koda, a označeni su da ne predstavjaju (*false negative*).

Ipak, ima par nedostatka. Svi projekti koje sadrži su napisani u *Java* programskom jeziku. *Java* je najzastupljeniji programski jezik za detekciju indikatora loše dizajniranog koda. Pored toga, skup podataka nije uravnotežen. To znači da postoji veliki broj instanci koje nisu označene da predstavljaju indikator loše dizajniranog koda, od onih što su označene da jesu. U tabeli 1 prikazana je, na primeru projekta *Apache Cassandra*, zastupljenost indikatora loše dizajniranog koda *feature envy* u odnosu na verziju projekta. Broj označenih instanci varira u odnosu na projekat. U nekim projektima ne postoji nijedna instanca koja je označena kao indikator loše dizajniranog koda.

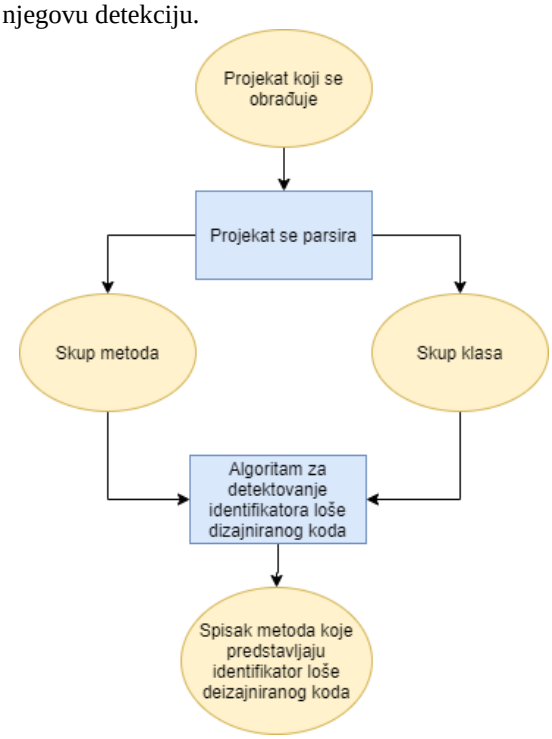
Tabela 1. *Zastupljenost indikatora loše dizajniranog koda po verziji projekta*

| Verzija | Broj klasa | Broj metoda | Broj linija koda | Zastupljenost *feature envy* |
|---|---|---|---|---|
| apache-cassandra-0.2 | 305 | 2511 | 69809 | 0 |
| apache-cassandra-0.3 | 302 | 2302 | 50040 | 2 |
| apache-cassandra-0.4 | 237 | 1857 | 36140 | 1 |
| apache-cassandra-0.5 | 239 | 1930 | 38471 | 163 |
| apache-cassandra-0.6 | 261 | 2143 | 42265 | 104 |
| apache-cassandra-0.7 | 376 | 3454 | 68490 | 381 |
| apache-cassandra-0.7.2 | 372 | 3300 | 66077 | 373 |
| apache-cassandra-0.7.3 | 376 | 3349 | 66895 | 0 |
| apache-cassandra-0.8 | 438 | 4226 | 81799 | 452 |
| apache-cassandra-0.8.1 | 432 | 4078 | 78546 | 76 |
| apache-cassandra-0.8.3 | 440 | 4176 | 80788 | 0 |
| apache-cassandra-1.0 | 506 | 4876 | 93333 | 0 |
| apache-cassandra-1.1 | 586 | 5730 | 110712 | 128 |

### 3.2. Arhitektura sistema

Na slici 1 predstavljen je opšti model sistema. Ulaz u sistem predstavlja projekat za koji se detektuju indikatori loše dizajniranog koda. Projekat se, zatim, parsira i iz njega se izvlači skup metoda i skup klasa zajedno sa njihovim sadržajem u obliku prirodnog jezika. Ti skupovi se prosleđuju algoritmu za detektovanje indikatora loše dizajniranog koda, koji na izlazu pruža spisak svih delova koda koji predstavljaju indikatore loše dizajniranog koda. Obzirom na to da je u ovom radu studija slučaja rađena za

*feature envy* indikator, na slikama je prikazan algoritam za njegovu detekciju.




Slika 1. *Opšti model sistema*

Algoritam za detektovanje identifikatora loše dizajniranog koda prolazi kroz prosleđen skup metoda i za svaku metodu izvlači skup klasa čije metode i polja koristi. Zatim računa sličnost metode sa klasom u kojoj se ona nalazi i sa klasama čije metode i polja koristi posmatrana metoda.

Posebno se beleži sličnost klase u kojoj se originalno nalazi metoda od ostalih klasa. Na kraju, proverava da li je najveća sličnost između metode i originalne klase, ili postoji klasa sa kojom je metoda koja se proverava sličnija.

Ukoliko takva klasa ne postoji, u okviru metode nije detektovan indikator loše dizajniranog koda. U suprotnom, metoda se predstavlja kao pozitivna instanca i rešenje bi bilo prebaciti tu metodu u onu klasu kojoj je ona najsličnija.

Pretprocesiranje podataka je vršeno na sledeći način:

- Sadržaj posmatranog dela koda se rastavlja na niz tokena.
- Iz niza tokena izbacuju se specijalni karakteri, višak praznih redova, kao i sve ključne reči programskog jezika - za Java programski jezik, ključne reči bi bile *class*, *private*, *public*, *if*, *else*, itd.
- Svaka reč iz niza se kasnije svodi na svoj koren reči (eng. *stemming*), koja se kasnije prosleđuje algoritmu za reprezentaciju teksta.
- Izlaz iz algoritma za reprezentaciju je matrica dimenzija [x,y], gde x predstavlja dimenziju vektora reprezentacije, a y predstavlja broj tokena koji je prosleđen na ulazu.

U radu su korišćena dva algoritma za reprezentaciju teksta:

- *Word2Vec* je statistički model za reprezentaciju reči. Algoritam koristi neuronsku mrežu za učenje veza između reči u okiviru prosleđenog korpusa. Ulaz u ovaj algoritam predstavlja skup reči na osnovu kojih se on obučava i nakon toga može da reprezentuje reči pomoću vektora realnih brojeva
- *BERT* je sekvencijalni model za reprezentaciju reči. Dizajniran je da trenira bidirekcionu reprezentaciju na osnovu nelabeliranog teksta [24]. Pretrenirani *BERT* model se može doučavati (eng. *fine-tune*) u zavisnosti od slučaja upotrebe. Za *BERT* algoritam neophodno je proslediti sekvencu reči. U zavisnosti od dužine sekvence, algoritam vriše transformaciju na različiti način. Treba naći optimalnu dužinu, tako da algoritam radi zadovoljavajuće. U radu je korišćena sekvenca od 2 linije koda.

## 4. REZULTATI I DISKUSIJA

Kako bi se metodologija evaluirala korišćene su mere: f-mera (*F-measure*), tačnost (*accuracy*) i odziv (*recall*). Rezultati su poređeni sa rezultatima *DECOR* i *TACO* alata za prepoznavanje indikatora loše dizajniranog koda. Obzirom na to da su korišćeni pretrenirani modeli algoritama za reprezentaciju reči, evaluacija je vršena direktno skupom podataka. Iz razloga što je vreme izvršavanja evaluacije dugačko zbog velike količine podataka i ograničenog vremena, iz celog skupa podataka izvučene su određene instance projekata nad kojima je vršen eksperiment. Instance su odabrane na takav način da svaka instanca projekta sadrži metode koje su označene kao indikator loše dizajniranog koda, dok rasprostranjenost indikatora varira od 2 metode do 10 po projektu.

Iz razloga što je *TACO* alat implementiran kao ekstenzija (eng. *plug-in*) koji je namenjen da se ugradi u okviru *Intellij* razvojnog okruženja, nije moguće automatski evaluirati ovaj alata i porediti ga sa rezultatima dobijenim evaluacijom sistema prezentovanog u ovom radu. Iz tog razloga, urađena je reimplementacija *TACO* alata. Iako je u radu [9] dobijena F-mera od 74% nad skupom podataka koji su koristili, u ovom radu nisu uspešno rekreirani ovi rezultati. Dobijena F-mera u ovom radu nad reimplementacijom *TACO* alata iznosi 15%.

U tabeli su prikazani rezultati dobijeni korišćenjem alata prezentovanog u ovom radu.

Tabela 2. *Rezultati dobijeni evaluacijom sistema*

| | Preciznost | Odziv | F-mera |
|---|---|---|---|
| *TACO* | 10% | 33% | 15.34% |
| *word2vec* algoritam za reprezentaciju | 9% | 50% | 15,25% |
| *BERT* algoritam za reprezentaciju | 9,09% | 50% | 15.38% |

Nakon analize rezultata, pronađeni su mogući razlozi za loše rezultate, kao i način na koji bi se mogli ukloniti i na taj način unaprediti sistem:

- Korišćen je parser koji je implementiran u okviru *RepositoryMiner* [2]alata. Obzirom na to da je ovo jedini deo koji je menjan u odnosu na originalnu implementaciju *TACO* alata, postoji verovatnoća da parser nije implementiran na ispravan način. Ukoliko parser ne radi ispravno, podaci koji se šalju algoritmu za detektovanje su neispravni i iz tog razloga nemoguće je dobiti dobre rezultate.

[2] https://github.com/antoineBarbez/RepositoryMiner.git

- Klasa u kojoj se nalazi metoda za koju se detektuje indikator loše dizajniranog koda kada se poredi sa metodom sadrži u telu metodu sa kojom se poredi. Na taj način, sličnost između klase i metode raste, jer se metoda poredi sama sa sobom u okviru klase. Potrebno je iz tela klase izbaciti posmatranu metodu i onda vršiti poređenje sa ostatkom klase, kako bi poređenje bilo validno.

## 5. ZAKLJUČAK

U ovom radu predstavljen je model koji omogućava automatsku detekciju indikatora loše dizajniranog koda. Za evaluaciju modela korišćen je skup podataka predstavljen u radu [10]. Detektovanje je vršeno korišćenjem informacija ekstrahovanim iz tekstualnog sadržaja – koda aplikacije. Za dobijanje informacija korišćeni su identifikatori promenljivih i komentari. Za reprezentaciju teksta korišćeni su *word2vec* i *BERT* algoritmi. Za računanje sličnosti između posmatranih delova koda korišćena je kosinusna sličnost.

Rezultati koji su dobijeni nisu zadovoljavajući. Problem može predstavljati implementacija drugih alata koji su korišćeni u okviru sistema. Neophodno je izvršiti validaciju korišćenih alata i ispraviti moguće greške ukoliko postoje. U daljem istraživanju biće izvršeno testiranje ovih alata, eksperimentsati sa drugim algoritmima za računanje sličnosti između vektora, kao i isprobane druge tehnike koje se koriste u obradi prirodnog jezika kako bi se rekreirali rezultati i na drugim skupovima podataka.

## 6. LITERATURA


[1] Martin, R.C., 2009. Clean code: a handbook of agile software craftsmanship. Pearson Education.

[2] Martin, R.C., 2002. Agile software development: principles, patterns, and practices. Prentice Hall.

[3] Fernandes, E., Oliveira, J., Vale, G., Paiva, T. and Figueiredo, E., 2016, June. A review-based comparative study of bad smell detection tools. In Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (p. 18). ACM.

[4] Alkharabsheh, K., Crespo, Y., Manso, E. and Taboada, J.A., 2019. Software Design Smell Detection: a systematic mapping study. Software Quality Journal, 27(3), pp.1069-1148.

[5] Dietz, L.W., Manner, J., Harrer, S. and Lenhard, J., 2018. Teaching clean code. In Proceedings of the 1st Workshop on Innovative Software Engineering Education.

[6] Fowler, M., 2018. Refactoring: improving the design of existing code. Addison-Wesley Professional.

[7] Azeem, M.I., Palomba, F., Shi, L. and Wang, Q., 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Information and Software Technology.

[8] Di Nucci, D., Palomba, F., Tamburri, D.A., Serebrenik, A. and De Lucia, A., 2018, March. Detecting code smells using machine learning techniques: are we there yet?. In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 612-621). IEEE.

[9] Palomba, F., Panichella, A., De Lucia, A., Oliveto, R. and Zaidman, A., 2016, May. A textual-based technique for smell detection. In 2016 IEEE 24th international conference on program comprehension (ICPC) (pp. 1-10). IEEE.

[10] Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R. and De Lucia, A., 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empirical Software Engineering, 23(3), pp.1188-1221

[11] Moha, N., Gueheneuc, Y.-G., Duchien, L., & Le Meur, A.-F. (2010). DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Transactions on Software Engineering, 36(1), 20–36. https://doi.org/10.1109/tse.2009.50


### Kratka biografija:


**Katarina-Glorija Grujić** rođena je 1998. godine. Osnovne akademske studije završila je 2019. godine na Fakultetu tehničkih nauka, na kom brani i master rad 2020. godine iz oblasti Elektrotehnike i računarstva – Računarstvo i automatika.
kontakt: katarina.glorija@uns.ac.rs