ITS za razvoj veština pisanja održivog koda

I. Problem

Inženjeri softvera treba da pišu *kvalitetan* kod koji ispunjava funkcionalne zahteve. Važna karakteristika *kvaliteta* koda je *održivost*. Održivost određuje lakoću razumevanja, izmene, proširivanja i testiranja softverskih komponenti [1]. Kod niske održivosti uvećava cenu razvoja softvera i loše utiče na ostale aspekte njegovog kvaliteta, poput bezbednosti, performansi i pouzdanosti [2][3]. Stoga industrija i akademija ističu da je važno da inženjeri softvera nauče da pišu održiv kod [4][5][6][7].

Postoji mnoštvo heuristika za pisanje održivog koda [8][9][10]. Mnoge su neprecizni opisi podložni subjektivnim interpretacijama [11] ili su previše tesno vezane za programske jezike, tehnološke konvencije ili arhitekturalne šablone [12]. Posledično, veštine pisanja održivog koda se stiču kroz izradu velikog broja zadataka uz podršku eksperta koji daje povratne informacije. Međutim, postoji oskudica eksperata koji bi mogli predavati ovo gradivo [13].

Inteligentni Tutoring Sistemi (*Intelligent Tutoring Systems*, ITS) bi mogli umanjiti ovaj problem simulacijom ljudskog instruktora. Ovaj rad predstavlja ITS specijalizovan za učenje pisanje održivog koda. Glavna komponenta ovog ITS-a je zadužena da prima kod od učenika i vraća personalizovane povratne informacije za ispravku tog koda.

Razvijeni ITS može da pomogne učenicima da savladaju osnove veštine pisanja održivog koda kroz manje izazove. Zahvaljujući tome, instruktori mogu posvetiti više vremena radu sa učenicima na sofisticiranijim problemima. ITS takođe podržava instruktore u otkrivanju miskoncepcija u znanju učenika.

II. Teorijske osnove

Koncepti koji bi morali biti opisani u teorijskim osnovama za razumevanje rešenja:

Šta predstavlja KLI radni okvir, zbog čega je on odabran i njegove komponente:

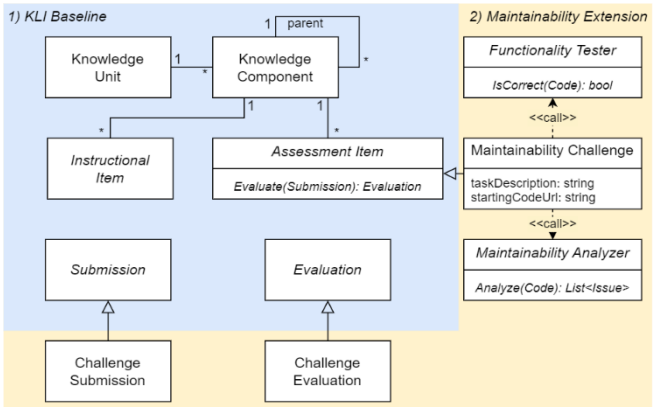
- *Instructiona items* – prezentacija edukativnih materijala koja uzrokuje porast znanja učenika (na primer, video ili tekst koji izlažu gradivo). *Instructional items* razvijaju nove i rafiniraju postojeće *Knowledge Components*
- *Knowledge Component* – veštine koje učenici mogu da primene da reše zadatke
- Zadaci su modelovani kroz *Assessment items* (na primer, odgovor na pitanje, kod koji je učenik predao kao rešenje).
- *Assessment items* evaluiraju (*Evaluate*) učenički odgovor (*Submission*)

Detekcija problema održivosti koda:

- Metrike koda i postupci u kojima se koriste

III. Rešenje

Slika 1 predstavlja domenski model rešenja. Osnova domenskog modela je *Knowledge-Learning-Instruction* (KLI) radni okvir predstavljen u poglavlju III.A. Nad ovom osnovom je izgrađena *Maintainabillity Challege* komponenta ITS-a (poglavlje III.B) koja je fokus ovog rada. Ova komponenta je specijalizovana za proveru koliko je učenika vešt u pisanju održivog koda.



Slika 1 Domenski model inteligentnog tutoring sistema za razvoj veštine pisanja održivog koda.

A. Knowledge-Learning-Instruction (KLI) radni okvir

Koncepti KLI radnog okvira su prikazani na segmentu 1) slike 1.

Svaki tip zadatka se izvodi iz entiteta *AssesmentItem* i implementira funkciju *Evaluate*. Svaki zadatak sadrži logiku koja:

1. prima odgovor (*Submission*)
2. proverava tačnost odgovora i, prema tome, određuje odgovarajuće povratne informacije (*feedback*)
3. vraća tačnost odgovora i povratne informacije kao evaluaciju (*Evaluation*).

Da bismo podržali specijalizovanu *Maintainability Challenge* komponentu, iz *Submission* i *Evaluation* klasa smo izveli klase *ChallengeSubmission* i *ChallengeEvaluation*. Ove klase enkapsuliraju specifičan ulaz koga očekujemo od učenika i odgovarajući izlaz koji vraća sistem.

Svaki specijalizovan zadatak zahteva i instrument (*learning instrument*) sa kojim učenik interaguje. Instrument je deo ITS-ovog korisničkog interfejsa.

B. Maintainabillity Challege komponenta

Izazov održivosti (*Maintainabillity Challege*) je specijalizovan tip zadatka za razvoj veština analize održivosti i refaktorisanja koda. Prikazan je u okviru segmenta 2) slike 1. Učenici predaju *rešenje izazova* (*challenge submission*), koje *izazov održivosti* evaluira da bi proizveo evaluaciju izazova (*challenge evaluation*).

Svaki izazov opisuje *zadatak* koji učenik treba da uradi. Primeri zadataka su:

- implementacija funkcije,
- promena ponašanja postojeće funkcije
- refaktorisanje funkcije da bi se njena održivost povećala, pri čemu se mora voditi računa da njena funkcionalnost ne bude izmenjena.

Izazov može da pruži *početni kod.* Početni kod može biti prazna funkcija, klasa sa delimično implementiranom logikom ili veliki projekat sa mnogo paketa i klasa. Ova varijabilnost omogućava instruktorima da podese obim zadataka – od ilustrativnog izazova za početnike, do realnih primera softverskih projekata koji mogu biti analizirani po pitanju održivosti i refaktorisani.

Pošto izazov zahteva da učenici kreiraju ili modifikuju kod, uveli smo tester funkcionalnosti (*functionality tester*) koji

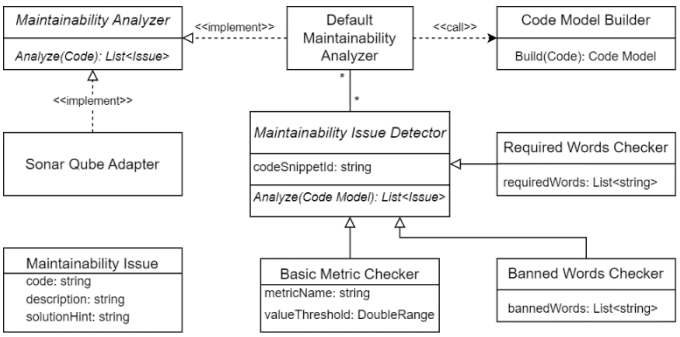
određuje da li je kod funkcionalno korektan. Tester funkcionalnosti nad kodom pušta automatizovane testove. Predstavlja značajan element procene održivosti koda, jer je kod koji nije funkcionalno korektan neadekvatan, bez obzira na svoj kvalitet [6].

Kod koji je učenik predao se analizira od strane *Maintainability Analyzer* komponente (slika 2):

1. Kod se procesira u model koda koji je pogodniji za analizu održivosti koda. Logika za procesiranje koda je enkapsulirana u (*Code Model Builder*) klasi. Logika procesiranja koda u model koda je tesno vezana za programski jezik, dok je model koda donekle nezavisan[1] od programskog jezika.
2. Na model koda se primenjuje jedan ili više detektora problema održivosti (*Maintainability Issue Detector*). Svaki detektor problema održivosti je primenljiv na ceo predati kod ili na deo predatog koda, definisan kroz *codeSnippetID* identifikator. Ovaj identifikator se poklapa sa potpisom klase (*class namespace* ii me klase) ili punim potpisom metode (koji obuhvata potpis klase i metode).

Dizajnirali smo tri tipa detektora problema održivosti:

1. *Basic Metric Checker* koji poredi vrednosti metrika koda sa predefinisanim pragovima. Primer ovakvog detektora je određivanje da li sve funkcije u kodu imaju između pet i 15 linija koda (metrika *Lines Of Code*, LOC)
2. *Banned Word Checker* osigurava da predefinisani skup neadekvatnih reči (*noise words* [5]) nije prisutan u predatom kodu. Na primer, ovakav detektor problema vrši proveru da li sva imena u predatom kodu ne sadrže generičke reči poput „the“, „data“, „info“, „int“, „list“ ili „var“.
3. *Required Word Checker* osigurava da predati kod sadrži imena iz predefinisanog skupa reči. Na primer, ovakav detektor bi proverio da li imena u kodu sadrže „DataRange“ i „Overlaps“.



Slika 2 Konceptualni model analize održivosti koda (*Maintainability Analyzer*).

NAPOMENA: poglavlje nije dovršeno. Primer ukazuje kako da opisujete svoj sistem. Obratite pažnju da tekst ne ponavlja imena i odnose klasa prikazanih na slikama, već daje dodatne informacije o tome kako je sistem dizajniran.

[1] Slični jezici (na primer, *Java* i C#) mogu biti konvertovani u isti model koda. Međutim, konvertovanje bitno različitih jezika (na primer, *JavaScript* i C) u zajednički model je upitno, pogotovo što se problemi održivosti u tom slučaju mogu značajno razlikovati.