Marija Bogavac 1068/23

Refaktorisanje klasa

U originalnom kodu postoje dve klase : Main.java i Sudoku.java.

1. Main.java – u njoj se nalaze metodi displayMenu, difficulty, parseAdd i play koga poziva main metod.
   - Metod difficulty sluzi da korisnik na standardnom ulazu unese koliko tesku sudoku zeli da resava.

   Ako ukuca broj 1 tezina je Easy,
   2 Medium,
   3 Hard,
   4 Expert,
   5 kraj igre i zavrsavanje main metoda sa statusom 0

   Ako je ukucan broj 1-4 onda je povratna vrednost ove funkcije 1,
   5 onda je povratna vrednost ove funkcije 0,
   bilo koji drugi broj return -1.

   Metod play zove duffuculty I ako je povratna vrednost 0 onda se izlazi iz play; ako je -1, ponoviti unos; ako je 1 onda se process nastavlja.

   Objekat usrSudoku je globalni za klasu Main. U zavisnosti od izabrane difficulty() se usrSudoku instancira u easy/medium/hard/expert Sudoku. Postoji i ispis sudoku table.

   Refaktorisanje:

   Iz Main klase, izvlacenjem difficulty metoda, nastala je klasa UserChoosingDifficulty. Ta klasa sadrzi samo taj metod i ima instancirane objekte za svaku vrstu sudoku-a. Ti objekti su staticki i difficulty metod je staticki. Prvo sam pokusala da pozovem difficulty metod (poziv je u play metodu) sa difficulty(sudoku), sudoku je null u play. Sudoku ce ostati null nakon poziva ove funkcije, sto nije zeljeno ponasanje, jer se prosledio parameter by value, menja se u difficulty ali kad se vratimo u play nece biti promenjen. Zato sam napravila wrapping klasu oko objekta Sudoku koji u sebi sadrzi samo konstruktor. Promanila sam poziv u

difficulty(usrSudokuWrapper). Wrapper sadrzi referncu objekta sudoku. Metod koji prihvata wrapper objekta menja Sudoku objekat unutar sebe da postane novi Sudoku objekat. Bez wrappera ako pokusamo da promenimo referencu na Sudoku objekat unutar metoda, ta promena nece biti vidljiva izvan metoda. Ovo je mozda moglo biti reseno i dodavanjem settera, tj. setBoard.

Prilikom testiranja nastao je **problem sa scanner**-ima. U play se instancira jedan scanner, poziva se difficulty koja instancira svoj scanner i parseAdd koja takodje instancira svoj scanner. Kada u testu uradim setIn (tj. setUpInput) za play metod onda dolazi do greske. Prvi scanner prodje a drugi izbacuje exception NoSuchElement jer nema sta njegov scanner.next() da procita. Pogresno je imati dvaput Scanner nad istim Stream-om jer jedan Scanner proguta ceo Stream. Ako pokrenemo aplikaciju sa Run (iz Main-a), unesemo vrednost 1, pritisnemo Enter, tada Scanner.next() poziv I dalje uzima sve sa ulaza ali na ulazu nemamo vise od 1\n. A kada vestacki (preko testa) prosledimo vise linija onda dolazi do greske jer Scanner.next() procita sve linije, a trebalo je samo jednu.

Zato je potrebno samo jednom instancirati Scanner (u play) I prosledjivati ga kao parameter metodima difficulty i parseAdd.

Takodje, metod difficulty sam refaktorisala zbog ponavljanja koda u svakom if-else. Dodala sam poslednji if(result==1){ponavljani kod}.

- Metod parseAdd sluzi za korisnicki unos polja (provera da li je dobro uneto polje A-I , 1-9) i vrednosti koju zeli da upise u polje (takodje provera vrednosti koja se upisuje 1-9). Ako je unos validan onda se vrednost upisuje u to polje metodom add,ispisuje se poruka kao I sudoku board sa dodatom novom vrednoscu i vraca se true. else u ovom metodu je nepotreban jer se svakako vraca false na kraju metode.

Refaktorisanje: Iz Main klase, izvlacenjem parseAdd metoda, nastala je klasa UserTypeInField. Zbog gore pomenutog problema sa scannerom kao i nacina prosledjivanja parametara (takodje

opisano gore) poziv ovog metoda iz play je parseAdd(usrSudokuWrapper,location,scanner);

- Metod play ostaje u klasi Main i nakon mog refaktorisanja. U njemu, nakon biranja difficulty Sudoku poziva se metod displayMenu koji daje mogucnost korisniku da izbare sledece akcije :

Ako ukuca 1 onda treba da ukuca polje u koje zeli da upise neku vrednost. Ako metod parseAdd vrati false onda se vrednost ne moze upisati i treba ponovo izabrati akciju. Inace u undoList se doda uneseno polje (undoList sluzi za 2.akciju)

Ako ukuca 2 onda bira da Ponisti poslednji potez. Iz objekta sudoku se izbacuje vrednost iz poslednje-unetog polja putem remove metoda i iz undoList se izbacuje poslednji element. Ovde postoji provera da li je polje ispravno uneseno ali za to nema potrebe jer je parseAdd to vec proverio : sled dogadjaja je da uvek prvo unesemo polje, proverimo ga u parseAdd a zatim izbacimo iz Sudoku objekta. Moji testovi uvek postizu if(true), nikad if(false) zbog objasnjenja iznad u 53 liniji

if (obj.matches("[A-I][1-9]")).

Jedan moj test proverava da li je moguce uraditi 2. akciju a da uopste nismo uneli ni jedan element. Ne moze zbog promenljive index.

Ako ukuca 3 onda se pozove solve metod koji resi celu sudoku. Nacin resavanja i njegove mane objasnicu kasnije.

Ako ukuca 4 onda odustaje od Sudoku sa tim difficulty levelom, izlazi se iz unutrasnje petlje a spoljasnja ce opet prikazati izbor difficulty levela.

Ako ukuca bilo sta drugo pojavice se poruka da je los unos i treba pokusati ponovo.

2. Sudoku.java

Ova klasa sadrzi privatne nizove i matrice koji pomazu pri analiziranju da li je ispravan red, kolona ili celija (ja cu zvati box, autor originalnog koda zove Cell a polja zove Blocks sto je zbunjujuce po meni – podgledati deo lose imenovanje).

Sudoku konstruktor pravi Sudoku objekat od vrednosti incijalne table koje su njemu prosledjene i to pozivanjem funkcije parseString koja upisuje u board od tog objekta vrednosti redom po boxovima. Konstruktor je ovde protected a u mojoj refaktorisanoj verziji je public jer mi je potreban u paketu tests.

Ima dosta metoda i neke se koriste za prikazivanje Sudoku table, a neke su za logiku resavanja i tumacenja table. Zato sam ja odvojila u dve klase: istoimena Sudoku i SudokuDisplay.

Polja u **SudokuDisplay** su board i usrBoard. Konstruktor SudokuDisplay se sastoji od dva get poziva: getBoard i getUsrBoard. getUsrBoard je dodat zbog refaktorisanja klase Sudoku na dve klase Sudoku i SudokuDisplay. Ta dva get metoda se pozivaju u konstruktoru objekta SudokuDisplay. Drugo resenje bi bilo da su u SudokuDisplay staticke metode I kao parametar primaju Sudoku objekat, pa njega koriste da naprave display tj. ispis table.

Ali sta su board i usrBoard?

Board je 9x9 matrica koja sadrzi inicijalne vrednosti za easy, medium, hard ili expert sudoku objekta. usrBoard, takodje 9x9 matrica, na pocetku programa je uvek ispunjen nulama. Kako korisnik unosi vrednosti u polja, tako se te vrednosti unose u usrBoard. Npr ako korisnik na A1 polje unese broj 3, onda ce usrBoard na poziciji (0,0) imati upisan broj 3.

Autor originalnog koda kaze:

“Throughout the entire game, the original board is not changed, all the changes happen in the usrboard Sudoku board. This is because if

the user is filling the value and it alters the original board, then the computer wouldn't be able to figure out which block is removable and which one is not. If the user accidentally removed a default block that is not supposed to be removable, that could drastically change the outcome of the game and maybe even making the computer unable to solve it".

Ja bih to drugacije uradila.

Ideja 1: postoji matrica board popunjena inicijalnim vrednostima u koju ce se upisivati vrednosti koje korisnik unosi, ali sa strane cuvamo I Boolean matricu boolBoard koja odgovara board matrici u smislu: gde je 0 u board, u boolBoard je false. Gde su 1-9 u board, u boolBoard je true. Ako treba da se izbrise neki element npr. u polju C2 onda se trazi u boolBoard da li je ta vrednost false, ako jeste moze se izbrisati vrednost na poziciji (2,1). Elementi u boolBoard se ne menjaju.

Ideja 2 : koriscenjem hash mape. Key u mapi bi bio string polje, a value u mapi par vrednost I indicator da li je ta vrednost inicijalna ili ju je uneo korisnik.

Kad bismo resili da koristimo jedan Sudoku board sa vrednostima, onda ne bi bilo potrebe za nekim funkcijama I dupliranja u kodu (npr. toString i output – pogledaj deo lose imenovanje).

U SudokuDisplay su metode toString, output, getBoardAsString i funkcija repeat koja sluzi za lepsi prikaz sudoku sa crticama koje odvajaju boxove.

- Metod toString sluzi za prikaz table. Formatira je zasnovano na nizu board direktno, menjajuci 0 sa – i jos dodaje crtice za lepsi prikaz boxova (ovde se zove repeat).
- Metod output, vrlo slican metodu toString, formatira tablu tako sto proverava dva niza usrBoard i board. Takodje zamenjuje 0 sa -.
- Metod getBoardAsString ima suprotni posao od metode parseString (koja se poziva u konstruktoru) i od Sudoku table

pravi string u kome su sve vrednosti iz sudoku. Radi samo sa board.

Metodi u klasi **Sudoku** su presentInRow, presentInCol, presentInCell, analyzeRow, analyzeCol, analyzeCell, updateAnalyzed, findValue, findBlock, findOnlyInCell, findOnlyInRow, findOnlyInCol, solve, isSolved, add, remove. Neki su privatni, a neki public. Pravi je bio izazov da se razume ovaj kod. Previse ponavljanja i zapetljanih funkcija.

- Metod add – dodaje u usrBoard vrednost koju je korisnik uneo za odredjeno polje samo ako ta vrednost ne postoji u koloni, redu i boxu. Zato se pozivaju privatne metode presentInCol/Row/Cell.
- Metod remove – brise iz usrBoard vrednost koja je u odredjenom
  (preko parametara poslatom) polju. Ako pokusa da se izbrise vec neko inicijalno polje to je spreceno if-om.
- Metod isSolved proverava da li je popunjena cela Sudoku tabla. U usrBoard se upisuju sve vrednosti iz board. Za svako polje razlicito od 0, povecava se counter. Ovde postoji jedna mana u kodu a to je da autor nije znao za postojanje else if(){} grane, vec samo else{if(){…}}. Ako counter nije 81 (9x9 polja) onda ova sudoku nije resena I metod vraca false. Ako jeste 81, vrsi se jedna provera po redovima/kolonama/boxovima da li je neko polje u tabli 0. Umesto te provere mogle su se koristiti vec napisane funkcije za istu svrhu, npr. presentInRow/Col/Cell.
- Metod presentInRow proverava da li prosledjena vrednost postoji u tom redu. Analogno za presentInCol, presentInCell.
- Metod analyzeRow upisuje u matricu analyzedRow true ako je na toj poziciji u board broj 1-9. Inace je inicijalizovana na false el. Analogno za analyzeCol, analyzeCell (za analyze cell se koristi trodimenzioni niz). One se pozivaju u solve metodu.

- Metod updateAnalyzed se poziva u toku solve metoda i za unesenu vrednost u board azuriraju se matrice analyzedRow/Col…
  analyzedRow : false na 2. poziciji ako u board u tom row-u nema broja 2
- FindOnlyInCell – iz solve se poziva za svaki box. i ide po poljima u bloku, tmp[0]-1.polje, tmp[0][1]- u 1. polje upisujemo 2. Moramo proveriti da li je moguce upisati na tom mestu 2 pozivanjem funkcija analyzeRow/Col/Cell. Zatim pozivamo metod findValue koji proverava da li je 2 jedini broj koji moze biti upisan u tu poziciju. Ako je odgovor pozitivan broj (to znaci da) onda se u board upisuje broj 2.
  solvedCounter- koliko polja je vec popunjeno u tom boxu
  doneBlocks[i]=true – i-to polje je uspesno popunjeno.
  isUpdate – da li je postavljen broj na praznom polju
  Sada nastupa deo koda za koji je autor napisao : *// Find the block for a certain value*.
  Ako je solvedCounter < 9, u znacenju nije bio popunjen ceo box vec smo morali da trazimo value sa findValue (tj. radimo algoritam gore opisan) onda za tu vrednost trazimo polje u koje bismo mogli da je upisemo – findBlock metod kome su parametri tmp, doneBlocks i value. Ako postoji tacno jedno polje u koje se moze upisati value onda vraca indeks tog polja. board se azurira.
  U odnosu na isUpdate se salje odgovor solve metodu, da li je uspesno popunjeno polje. U solve se nalazi polje unProcCell/Row/Col koji pamti koliko boxova je neuspesno analizirano. Uslov

```
 if (unProcCol == 9 && unProcCell == 9 && unProcRow == 9) {
            finish = true;
        }
```

sluzi za shvatanje da nema vise progresa i onda je potrebno zavrsiti petlju. Losa stvar ovde je sto nema obavestenja (print) da li je nesto bilo uspesno ili neuspesno.
Ovaj princip se ponavlja i za findOnlyInRow/Col. Ako se pozove solve nad vec resenom sudoku onda ce solvedCount pomoci da se izadje iz petlje.

Lose imenovanje

- parseAdd je los izbor imena koje ne objasnjava sta taj metod radi. Moglo je i skeniranje lokacije biti unutar ovog metoda. Bolji naziv : userTypingLocationAndValue
- toString i output rade previse slican posao da bi to bile dve odvojene metode. Njihov naziv je donekle neodredjen jer se ne zna sta hocemo da prikazemo njihovim pozivom. Moze da postoji i samo jedna funkcija od njih dve.
- add, remove, isSolved, solve, findValue mogu imati duze ime koje objasnjava sta se dodaje, sta se resave…
- cell je autoru 3x3 box koji se nalazi u matrici, njih ima 9. A block je pojedinacno polje u tabli. Ne bih nazvala taj pojam block vec field ili position.
- Koriscenje niza sa nazivom tmp…

1. alat : jUnit

Koriscen za pisanje jedinicnih testova. Testirala sam funkcije koje su public u svim klasama. Potrebno je dodati @Test pre svakog testa. Iz te biblioteke koriscene su funkcije assertTrue, assertFalse, assertEquals, assertThrows. Opis testova kasnije.

2. alat : Mockito

Mockito se obično koristi za imitiranje eksternih zavisnosti (npr. baza podataka, API-ja ili korisnickih unosa) tokom testiranja jedinica da bi se izbegli njihovi stvarni pozivi baza podataka i njihovi spoljni uticaji. Kada se

pozove mock(SomeClass.class) ili mock(SomeInterface.class), Mockito kreira lazni objekat koji se ponasa kao instanca SomeClass ili implementira SomeInterface.

U mojim testovima iz klase UserChoosingDifficultyTest koristi se mock metod zbog Scanner-a.

- mock objekat klase Scanner je kreiran korišćenjem mock metode.

- when(scanner.next()).thenReturn("1") specifizira da kada se pozove next() metoda mock skenera, treba da se vrati string "1".

- verify(scanner).next() osigurava da je next() metod laznog skenera pozvan tokom izvrsavanja metode.

Postoji i metod spy koji “spijunira” postojeci objekat I dozvoljava pozive pravih funkcija dok kontrolise druge metode. Mock se pravi od tipa klase, ne od konkretne instance.

3. alat: Java Flight Recorder

Java Flight Recorder je profajler integrisan u Java Virtuelnoj Masini i sluzi za dijagnostikovanje uskih grla a time I kako optimizovati kod, koje funkcije se zovu, koliko puta I koliko resursa trose.

Ovaj alat se pokrece tako sto izaberemo opciju Run with Java Flight Recorder. A rezultate mozemo videte klikom na jednu ikonicu koja oznacava Profiler.

Tada mozemo videti sekcije: Flame graphs, Call Trees, Method List i Events.

Flame graphs pomazu pri vizualnoj reprezentaciji koji metodi trose najvise CPU vremena. Svaki pravougaonik je jedan metod. Siva boja predstavlja native calls – Java metodi implementirani u drugom jeziku npr C/C++. Zuta boja predstavlja aplikacijske metode i biblioteke. Prelazeci preko pravougaonika moze se videte procenat koliko je vremena taj metod oduzeo I koliko je memorije zauzeo.

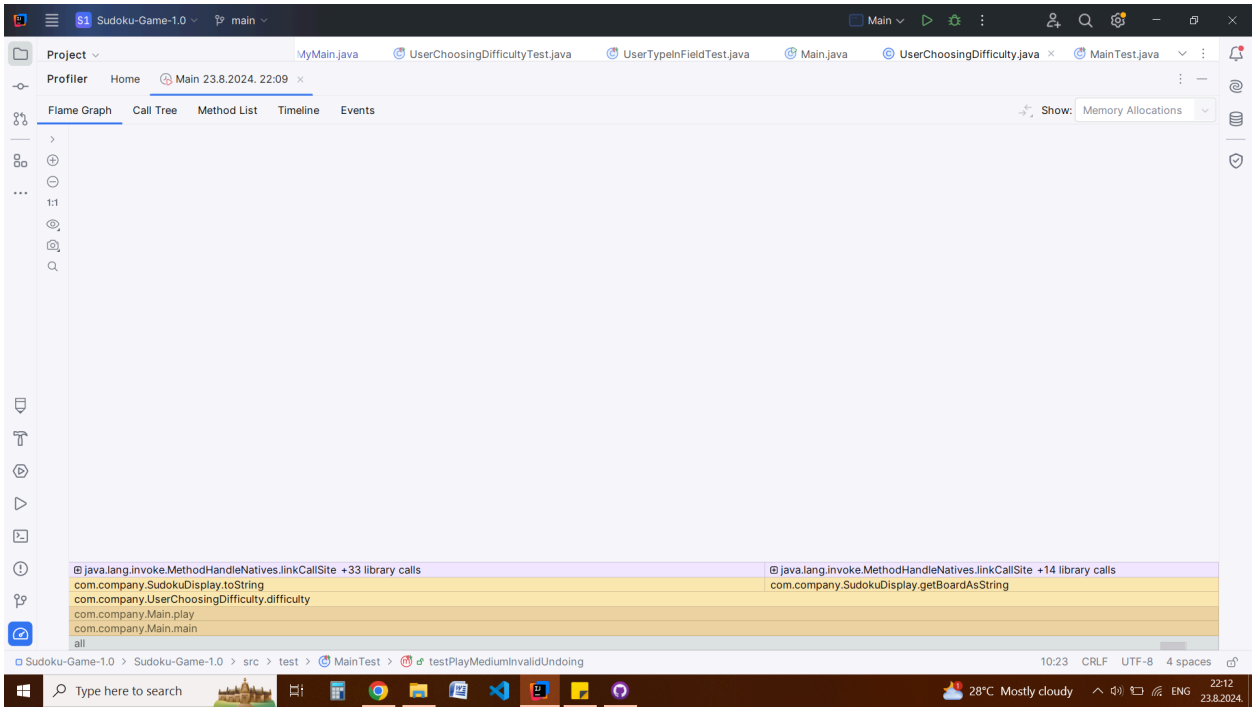
Call Trees su Flame Graphs u tekstualnoj formi

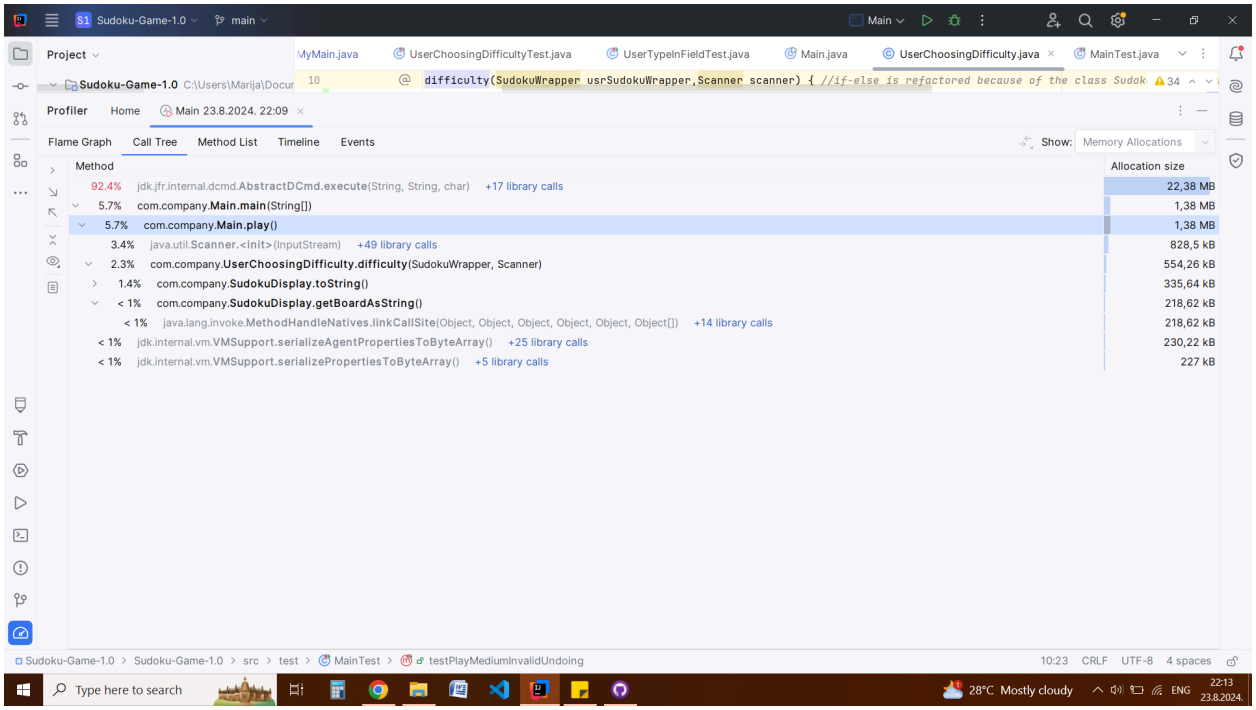
Method List je lista metoda koji se pokrecu, koliko puta. Pomaze pri identifikovanju metoda koji se najvise koriste i tako se moze optimizovati taj metod.

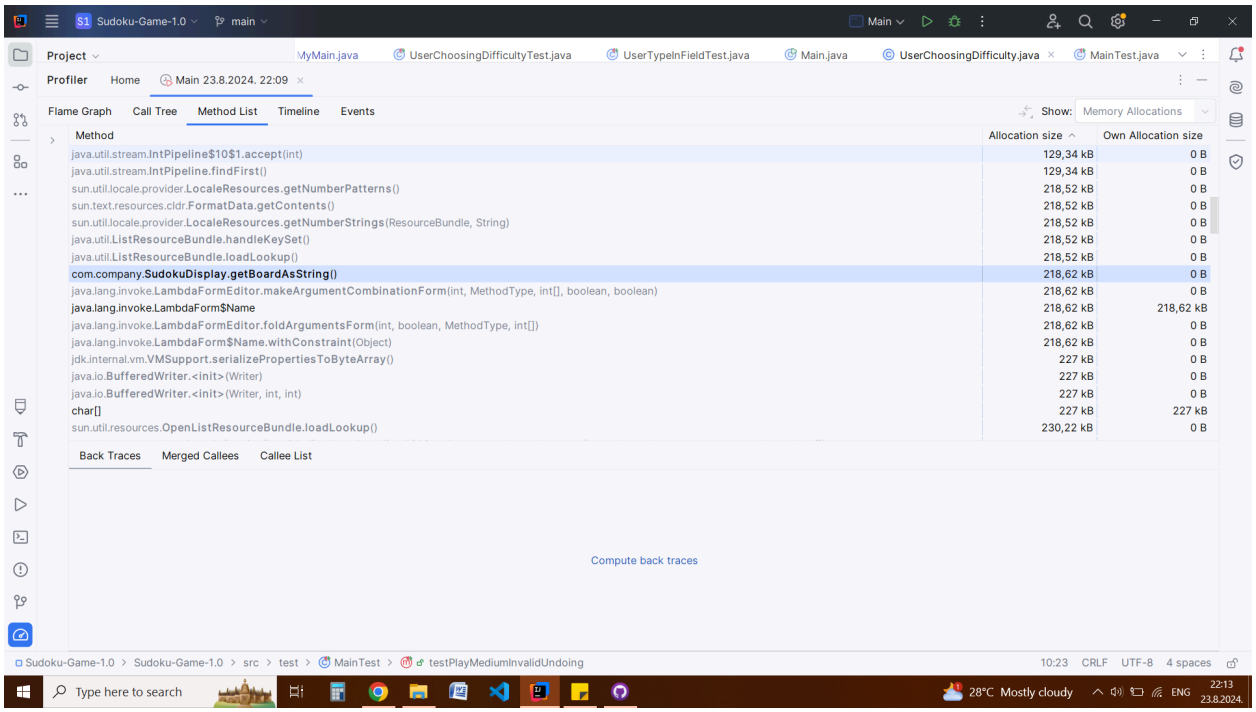
Events – podaci o npr. Garbage collection…

Vizualizacija podataka iz Recording.jfr je moguca preko Java Mission control (ali to moze samo za Eclipse okruzenje).

Pokrenula sam profajliranje za Main, upisala sam vrednost u polje.

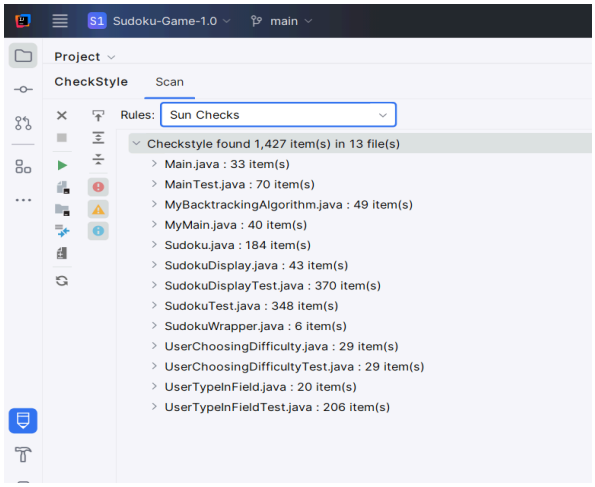






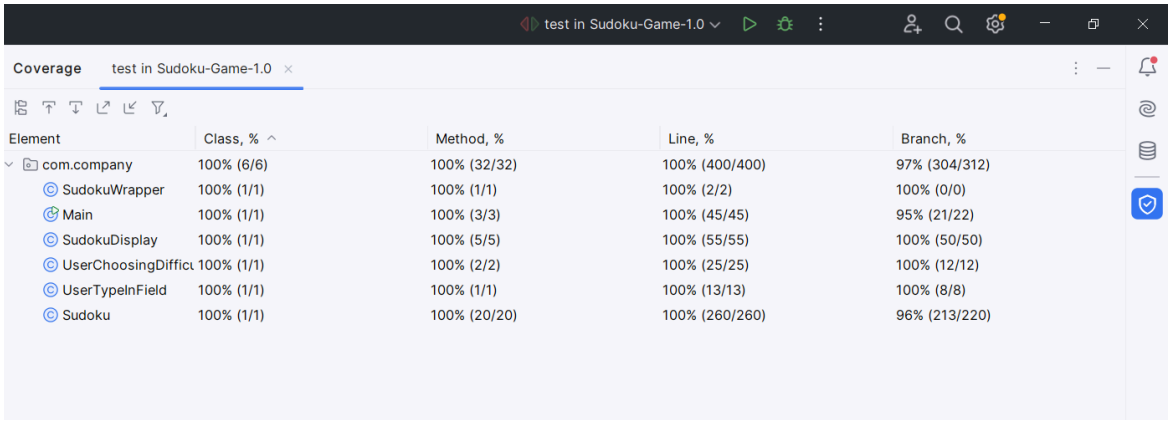
4. alat: CheckStyle

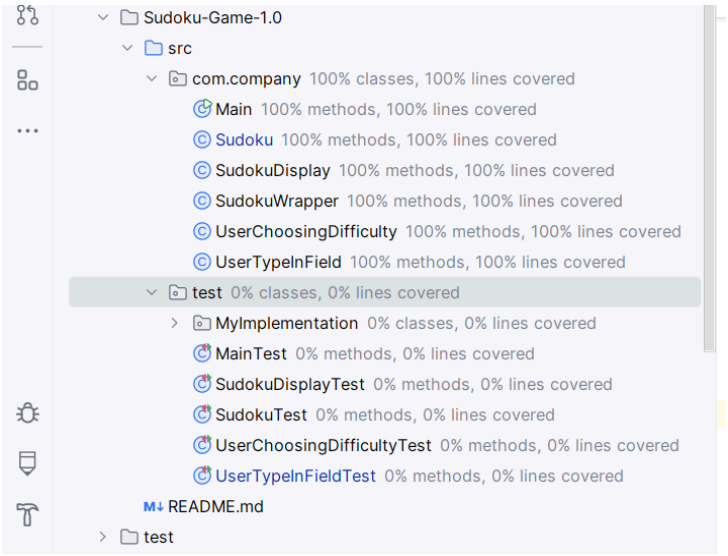
Alat za staticku analizu koda koji se koristi u razvoju softvera za proveru da li je Java izvorni kod uskladjen sa odredjenim pravilima kodiranja. Sun checks je skup standarda koje treba da proveri CheckStyle da bi se osiguralo primenjivanje konvencija i dobrih praksi u pisanju Java koda. Npr. imenovanje funkcija, klasa, maksimalna duzina linija, postovanje belina da bi kod bio citljiv.



5. alat: JaCoCo

Pokrivenost koda u IntelliJ IDEA omogućava da analiziramo koje su linije koda izvršene tokom određenog pokretanja. Pomaže u određivanju procenta koda pokrivenog testovima i identifikaciji oblasti kojima nedostaje pokrivenost testovima.
Kad se klikne run with code coverage, kod ce postati obojen trima bojama:
Zelena znaci da je taj deo koda pokriven
Crvena znaci da nijedan test nije ispitivao taj deo koda
Zuta znaci da je parcijalno testirano, npr. ako je to if-else onda se uslo samo kad je uslov true, ali ne I kad je false.
U mom kodu postoje false hits koji oznacavaju deo koji u stvari i nije skroz pokriven od strane testova. Npr. to se moze videti u liniji 108 u Sudoku.java
if (value > 0 && value <= 9) – ovde nisam isprobala za sve value od 1-9.
Code coverage svih klasa u projektu je 100%.





Testovi i zaključci:

U mojim test klasama postoje anotacije @BeforeEach tj. sta uraditi pre svakog testa – postaviti System.in i System.out i @AfterEach, vratiti originalni System.in i System.out.

1. MainTest – simuliram korisnicki unos, moze biti pogresan pokusaj undo, pogresna difficulty, pokusaj upisivanja vrednosti u polje koje je u inicijalnoj tabeli. Nisam primetila propuste, svi testovi prolaze. Testiram sa assertTrue po ispisu na standardni izlaz.
2. UserChoosingDifficultyTest – koristim Mockito da bih simulirala koriscenje Scanner-a

- mock objekat klase Scanner je kreiran korišćenjem mock metode.
- when(scanner.next()).thenReturn("1") specifizira da kada se pozove next() metoda mock skenera, treba da se vrati string "1".
- verify(scanner).next() osigurava da je next() metod lažnog skenera pozvan tokom izvršavanja metode.

Testiram invalid inpute za slovo, veliki broj, 0, negativan broj.
Nisam primetila propuste, svi testovi prolaze. Testiram sa assertEquals.

3. UserTypeInFieldTest – u svakom testu proveravam da li je moguce dodati u col i row odredjenu vrednost. Onda proveravam da li se u ispisu nalazi ocekivana Sudoku tabla. Testiram invalid inpute za vec zauzeta polja, los naziv polja, losu vrednost koja se unosi u polje... Nisam primetila propuste, svi testovi prolaze. Testiram sa assertThrows (za InputMismatchException), assertTrue, assertFalse, assertEquals.
4. SudokuDisplayTest – proveravam da li je ispis ocekivani sa assertEquals, gde ja sama postavljam kako bi trebalo da izgleda tabla. Nisam primetila propuste, svi testovi prolaze.
5. SudokuTest – dolazimo do test klase sa propustima. U kodu pored svakog neuspesnog testa napisan je razlog, ali ponovicu i ovde. Svi ovi testovi su u slucaju da pocetna sudoku nije easy/medium/hard/expert.
    1) testingConstructorAddedExtraFieldSudoku – ocekujem da se baci neki exception jer zelim da upisem 82 vrednosti (u obliku stringa) u 81 polje
    2) solveSudokuWithWrongLastRow – ocekujem da se baci neki exception jer je pogresan poslednji red, nema provere da li je dobra tabla
    3) solveMediumSudokuAddedValueInE3AttemptWithNumber1 – solve metod ne gleda korisnicki unos (tj. nigde u solve metodu nema pominjanja usrBoard). Ako u jednom polju mogu da se unesu dve vrednosti, recimo 6 i 1, korisnik uspesno unese 1 i zeli da prestane sa igrom i vidi resenje, ta 1 se nece nalaziti na

tom mestu jer je ipak 6 bio ispravan odgovor – to se moze videti u testu solveMediumSudokuAddedValueInE3AttemptWithNumber6

4) solveEmptySudoku – da li solve moze da resi praznu sudoku? Ne
   Ovaj metod sam dodala zarad potpune pokrivenosti jer nikad se ne moze uci u 401. liniju
   ```
   if (unProcCol == 9 && unProcCell == 9 && unProcRow == 9) {
           finish = true;
       }
   ```
5) solveRandomSudoku – ne ume da resi unapred zadatu tablu.

Bolji solver za Sudoku

U paketu test/MyImplementation nalazi se kako bih ja napravila solver za Sudoku koristeci backtracking umesto grube sile.

Backtracking algoritam pokušava da reši Sudoku tako što proverava da li svaka ćelija ima validno rešenje.

Kada nijedno ograničenje nije prekršeno, algoritam prelazi na sledeću ćeliju, popunjava sva moguća rešenja i vrši neophodne provere.

Ako je ograničenje prekršeno, algoritam povećava vrednost ćelije. Ako vrednost predje vrednost 9, algoritam se vraća na prethodnu ćeliju i menja njenu vrednost.

Detaljnije,

ide element po element (prvo prolazi kroz kolone, pa redove) i stavlja vrednost (od 1 do 9). Sa metodom isValid() proverava se da li se ta vrednost uklapa u taj red, kolonu i polje.

- Ako vrednost može biti u tom polju, ulazimo u rekurziju sa tablom sa tim poljem popunjenim.
- Ako vrednost ne može biti u tom polju, onda gledamo da li sledeća vrednost može. To radimo dok ne isprobamo svaku vrednost od 1 do 9 i

nijedna ne uspe. To znači da je neki prethodni korak (ili koraci) pre ovog trenutnog bio loš (na primer u neko polje možemo da stavimo 2 ili 6 i stavimo 2. To pokazuje da ne možemo da popunimo sledeće prazno polje u tom redu zbog lošeg izbora – trebalo je izabrati 6). Kada se to dogodi, pravimo korak unazad (vraćamo se na prethodno polje u redu) vraćanjem false, a zatim traženjem druge vrednosti u tom polju. Ako ne možemo ponovo pronaci vrednost, ponovo se vraćamo unazad na prethodno polje...

Ovaj metod moze da resi bilo koju sudoku, a originalni kod koji sam proucavala moze resiti samo sudoku sa jednim resenjem. Autor je vec zadao sudoku unapred, nema samostalnog generisanja table sto je propust. Pokretanjem igrice iznova i iznova resavaju se uvek iste sudoku table.

Testovi pokazuju da ovaj solver ne resava unapred zadate (random,empty) sudoku table koje sam ja unela u testu.