UNIVERZITET U NOVOM SADU
PRIRODNO-MATEMATIČKI FAKULTET
DEPARTMAN ZA
MATEMATIKU I INFORMATIKU

Stefan Jaćović

Refactoring - IntelliJ, Java

Novi Sad, 2024.

Refaktorisanje je ključni proces u razvoju softvera koji se odnosi na preuređivanje koda bez menjanja njegove spoljašnje funkcionalnosti, s ciljem poboljšanja njegove čitljivosti, održivosti i performansi. IntelliJ IDEA, jedan od najpopularnijih IDE-ova (integrisanih razvojnih okruženja), pruža moćne alate za refaktorisanje koji olakšavaju ovaj proces.

U ovom radu su korišćene sledeće ugrađene metode za refaktorisanje:

1. Encapsulating fields
2. Extract method
3. Rename method
4. Inline variable
5. Introduce variable
6. Move instance
7. Safe delete

Opisi nekih od opcija za refaktorisanja u IntelliJ IDEA:

1. **Rename (Preimenovanje):**

   Omogućava preimenovanje promenljivih, metoda, klasa, paketa i drugih elemenata. Refactoring Engine automatski ažurira sve reference na preimenovani element.

2. **Extract method (Ekstrakcija metode):**

   Omogućava ekstrakciju dela koda u novu metodu. Refactoring Engine generiše novu metodu i zamenjuje originalni kod pozivom nove metode.

3. **Change signature (Promena potpisa):**

   Omogućava promenu potpisa metode (npr. dodavanje ili uklanjanje parametara). Refactoring Engine ažurira sve pozive metode kako bi reflektovao novi potpis.

4. **Move (Premeštanje):**

   Omogućava premeštanje klasa, metoda ili polja u druge klase ili pakete. Refactoring Engine ažurira sve reference na premeštene elemente.

5. **Inline:**

   Omogućava zamenu poziva metode ili korišćenja promenljive njenom definicijom. Refactoring Engine automatski zamenjuje sve instance poziva ili korišćenja.

6. **Safe delete (Sigurno brisanje):**

   Omogućava sigurno brisanje koda uz proveru da li postoje referencije koje bi bile prekinute. Refactoring Engine osigurava da nema prekinutih referenci pre brisanja elementa.

7. **Introduce Variable/Constant/Field:**

   Omogućava kreiranje nove promenljive, konstante ili polja iz izraza. Refactoring Engine automatski generiše deklaraciju i zamenjuje sve instance izraza novom promenljivom, konstantom ili poljem.

8. **Encapsulate fields**

   Encapsulacija polja u IntelliJ IDEA pretvara javna polja u privatna i generiše odgovarajuće metode za pristup (getere) i promenu (setere) tih polja. Tako se poboljšava kontrola pristupa i održivost koda.

Interni mehanizmi refaktorisanja u IntelliJ IDEA

Ključni mehanizmi su: Abstract Syntax Tree (AST), Program Structure Interface (PSI) i Refactoring Engine.

Abstract Syntax Tree (AST) u IntelliJ IDEA

Abstract Syntax Tree (AST) je ključna struktura u analiziranju i manipulaciji izvornog koda. AST je hijerarhijsko stablo koje predstavlja apstraktnu sintaksu izvornog koda, gde svaki čvor u stablu odgovara određenom konstruktu jezika, kao što su klase, metode, izrazi, deklaracije promenljivih i drugi sintaksni elementi. Ovo stablo omogućava alatima da razumeju strukturu koda na dubljem nivou, omogućavajući operacije kao što su refaktorisanje, navigacija i analiza koda.

Interna Struktura AST-a

AST je organizovan na način koji reflektuje sintaksu i logičku strukturu izvornog koda.

Npr.

```
public class Example {
   public void method() {
      int x = 5 + 3;
   }
}
```

AST za ovaj kod izgleda ovako:

```
Class (Example)
├── Method (method)
│   ├── Declaration (int x)
│   │   └── Expression (5 + 3)
```

- **Class čvor** predstavlja klasu Example.
- **Method čvor** predstavlja metodu method.
- **Declaration čvor** predstavlja deklaraciju promenljive x.
- **Expression čvor** predstavlja izraz 5 + 3.

Svaki čvor u AST-u sadrži informacije o tipu sintaksnog elementa, njegovim svojstvima (kao što su imena i tipovi), i njegovim potomcima.

Upotreba AST-a u IntelliJ IDEA

1. **Parsiranje i analiza izvornog koda:**

   Parsiranje je proces pretvaranja izvornog koda u AST. IntelliJ IDEA koristi parsere specifične za svaki podržani programski jezik koji analiziraju izvorni kod i kreiraju odgovarajući AST. Ovo uključuje:

   - **Tokenizacija:** Razbijanje izvornog koda na osnovne sintaksne komponente (tokene).
   - **Sintaksna analiza:** Analiza tokena prema pravilima sintakse jezika i njihovo organizovanje u AST.

   Na primer, za jednostavan izraz int x = 5 + 3;, parser će kreirati AST sa čvorovima za deklaraciju promenljive i aritmetički izraz.

2. **Navigacija kroz kod i identifikacija strukturnih elemenata:**

AST omogućava IntelliJ IDEA da lako prolazi kroz strukturu koda. Svaki čvor u AST-u može biti pretražen i analiziran, što omogućava identifikaciju specifičnih elemenata koda. Na primer:

- Pronalaženje svih metoda u klasi.
- Identifikacija svih instanci upotrebe određene promenljive.
- Pretraga svih poziva određene metode.

3. **Omogućavanje preciznog refaktorisanja:**

   AST je ključan za precizno refaktorisanje koda jer omogućava IntelliJ IDEA da identifikuje i modifikuje relevantne delove koda bez narušavanja njegove funkcionalnosti. On to radi tako što izvršava sledeće:

   - **Identifikacija ciljanih čvorova:** Kada korisnik zatraži refaktorisanje (npr. Extract Method), IntelliJ koristi AST da identifikuje sve čvorove koji će biti pogođeni.
   - **Modifikacija AST-a:** IntelliJ vrši potrebne promene u AST-u, kao što su kreiranje novih čvorova za nove metode ili promena postojećih čvorova za preimenovanje elemenata.
   - **Generisanje modifikovanog koda:** Nakon što su promene izvršene u AST-u, IntelliJ generiše izmenjeni izvorni kod na osnovu novog AST-a.

Prednosti Korišćenja AST-a

- **Preciznost:** AST omogućava preciznu analizu i modifikaciju koda jer predstavlja sve strukturne elemente koda.
- **Bezbednost:** Korišćenjem AST-a, IntelliJ može osigurati da refaktorisanje ne uvodi greške u kod. Na primer, promena potpisa metode automatski ažurira sve pozive te metode.
- **Efikasnost:** AST omogućava efikasno pretraživanje i navigaciju kroz kod, što je posebno korisno za velike projekte.

Program Structure Interface (PSI) u IntelliJ IDEA

Program Structure Interface (PSI) je sloj apstrakcije iznad Abstract Syntax Tree (AST) koji omogućava manipulaciju kodom na višem nivou. PSI predstavlja strukturu koda kroz objekte koji su specifični za određene jezike, omogućavajući alatu kao što je IntelliJ IDEA da se lako kreće kroz kod i manipuliše njime na intuitivan način. PSI pruža način da se pristupi, pretražuje i menja struktura koda, što omogućava bogat skup funkcionalnosti kao što su automatsko kompletiranje (auto-complete), pretraga referenci i refaktorisanje.

Interna Struktura PSI-a

PSI je organizovan u hijerarhijsku strukturu koja reflektuje sintaksu i logičku strukturu izvornog koda, slično kao AST. Međutim, PSI pruža dodatne funkcionalnosti i apstrakcije koje olakšavaju rad sa kodom. PSI elementi predstavljaju različite delove koda, kao što su klase, metode, promenljive, izrazi, komentari i drugi sintaksni elementi.

```
public class Example {
   public void method() {
      int x = 5 + 3;
   }
}
```

PSI struktura za ovaj kod izgleda ovako:

- **PsiClass:** Predstavlja klasu Example.

- **PsiMethod:** Predstavlja metodu method.
- **PsiField:** Predstavlja deklaraciju promenljive x.
- **PsiExpression:** Predstavlja izraz 5 + 3.

Upotreba PSI-a u IntelliJ IDEA

1. **Omogućavanje naprednih funkcionalnosti:**

   PSI omogućava IntelliJ IDEA da pruži napredne funkcionalnosti kao što su auto-complete, pretraga referenci i navigacija kroz kod. Ovo se postiže kroz sledeće mehanizme:

   - **Auto-complete:** PSI omogućava IntelliJ-u da brzo pretražuje dostupne PSI elemente u trenutnom kontekstu i predlaže moguće dopune koda. Na primer, kada korisnik počne da piše ime promenljive ili metode, IntelliJ koristi PSI da pronađe sve moguće opcije i prikaže ih korisniku.
   - **Pretraga referenci:** IntelliJ može efikasno pretraživati sve instance korišćenja određenih PSI elemenata. Na primer, korisnik može pretražiti sve reference na određenu metodu ili promenljivu, a IntelliJ koristi PSI za brzo identifikovanje svih mesta u kodu gde se taj element koristi.
   - **Navigacija kroz kod:** PSI omogućava IntelliJ-u da pruži funkcionalnosti kao što su "Go to Definition" ili "Find Usages". Kada korisnik klikne na ime metode ili klase, IntelliJ koristi PSI da pronađe deklaraciju tog elementa i upućuje korisnika na odgovarajuću lokaciju u kodu.

2. **Manipulacija kodom tokom refaktorisanja:**

   PSI igra ključnu ulogu u refaktorisanje koda jer omogućava identifikaciju i ažuriranje svih instanci koje treba promeniti. Na primer:

   - **Identifikacija ciljanih elemenata:** Kada korisnik zatraži refaktorisanje (npr. Rename), IntelliJ koristi PSI da identifikuje sve PSI elemente koji su povezani sa ciljnim elementom. Na primer, ako korisnik želi da preimenuje promenljivu, IntelliJ koristi PSI da pronađe sve reference na tu promenljivu u celom projektu.
   - **Ažuriranje koda:** Nakon identifikacije, IntelliJ koristi PSI da ažurira sve instance ciljanog elementa. Na primer, pri preimenovanju metode, IntelliJ koristi PSI da ažurira sve pozive te metode kako bi se osiguralo da novi naziv metode bude pravilno reflektovan u celom kodu.

Prednosti Korišćenja PSI-a

- **Intuitivna manipulacija koda:** PSI omogućava manipulaciju kodom na višem nivou apstrakcije, što olakšava implementaciju naprednih funkcionalnosti kao što su auto-complete i refaktorisanje.
- **Efikasna pretraga i navigacija:** PSI omogućava brzo pretraživanje i navigaciju kroz kod, što je ključno za velike projekte.
- **Integracija sa drugim alatima:** PSI se integriše sa drugim alatima i mehanizmima u IntelliJ IDEA, omogućavajući složene operacije kao što su analiza koda i bezbedno refaktorisanje.

Refactoring Engine u IntelliJ IDEA

Refactoring Engine je centralni mehanizam u IntelliJ IDEA koji upravlja procesom refaktorisanja koda. Refactoring Engine koordinira sve faze refaktorisanja, uključujući analizu koda, identifikaciju zavisnosti, primenu promena i ažuriranje referenci.

Faze rada Refactoring Engine-a

1. **Analiza koda:**

   Prva faza rada Refactoring Engine-a je detaljna analiza koda. Ovo uključuje:

   - **Parsiranje izvornog koda:** U ovoj fazi, Refactoring Engine koristi parsere specifične za svaki jezik kako bi pretvorio izvorni kod u Abstract Syntax Tree (AST).
   - **Generisanje PSI-a:** Na osnovu AST-a, generiše se Program Structure Interface (PSI) koji predstavlja apstraktne elemente koda.
   - **Identifikacija strukturnih elemenata:** Analiziraju se PSI elementi kako bi se identifikovali strukturni elementi koda, kao što su klase, metode, promenljive i izrazi.
   - **Detekcija zavisnosti:** Tokom analize, Refactoring Engine identifikuje zavisnosti između različitih delova koda, kao što su pozivi metoda, nasleđivanje i korišćenje promenljivih.

2. **Identifikacija zavisnosti:**

   Pre primene promena, Refactoring Engine identifikuje sve zavisnosti koje bi mogle biti pogođene refaktorisanjem. Ova faza uključuje:

   - **Analiza međuzavisnosti:** Refactoring Engine analizira kako su različiti delovi koda međusobno povezani. Na primer, promena naziva metode može uticati na sve pozive te metode u projektu.
   - **Detekcija spoljnih zavisnosti:** Pored unutrašnjih zavisnosti, identifikuju se i spoljne zavisnosti, kao što su korišćenje klasa i metoda iz biblioteka trećih strana.
   - **Priprema za refaktorisanje:** Na osnovu identifikovanih zavisnosti, Refactoring Engine priprema strategiju za primenu promena, osiguravajući da sve zavisnosti budu pravilno ažurirane.

3. **Primena promena:**

   Nakon identifikacije zavisnosti, Refactoring Engine primenjuje promene na kodu. Ova faza uključuje:

   - **Modifikacija AST-a:** Refactoring Engine menja AST kako bi reflektovao tražene promene. Na primer, pri preimenovanju metode, AST se ažurira da koristi novi naziv metode.
   - **Ažuriranje PSI elemenata:** Nakon modifikacije AST-a, PSI se ažurira kako bi reflektovao promene. Ovo osigurava da svi apstraktni elementi koda budu ispravno modifikovani.
   - **Generisanje izmenjenog koda:** Na osnovu ažuriranog AST-a i PSI-a, generiše se izmenjeni izvorni kod.

4. **Ažuriranje referenci:**

   Poslednja faza refaktorisanja je ažuriranje svih referenci na promenjene elemente kako bi se osigurao integritet koda. Ovo uključuje:

   - **Pronalaženje referenci:** Refactoring Engine pretražuje ceo projekat kako bi pronašao sve reference na promenjene elemente. Na primer, sve pozive preimenovane metode.

- **Ažuriranje referenci:** Sve reference se ažuriraju kako bi koristile nove nazive ili strukture. Ovo osigurava da nema prekinutih referenci ili grešaka u kodu.
- **Validacija promena:** Nakon ažuriranja referenci, Refactoring Engine vrši validaciju koda kako bi se osiguralo da sve promene budu ispravno primenjene i da nema sintaksnih ili semantičkih grešaka.

Uz pomoć alata **PSI viewer** u InteliJ se može videti detaljan izgled PSI.

Primer kako PSI izgleda pre i nakon korišćenja **Extract Method** opcije u ovom projektu:

Pre korišćenja

- PsiMethod (printStudentDetails)
    - PsiParameter (studentId)
    - PsiIfStatement
        - PsiMethodCallExpression (findStudent)
        - PsiBlockStatement (if true block)
            - PsiExpressionStatement (System.out.println("Student Details:"))
        - PsiBlockStatement (else block)
            - PsiExpressionStatement (System.out.println("Student not found"))

Posle korišćenja

- PsiMethod (printStudentDetails)
    - PsiParameter (studentId)
    - PsiIfStatement
        - PsiMethodCallExpression (findStudent)
        - PsiExpressionStatement (extracted())
        - PsiBlockStatement (else block)
            - PsiExpressionStatement (System.out.println("Student not found”)')

PsiMethod (extracted)

- PsiBlockStatement
    - PsiExpressionStatement (System.out.println("Student Details:"))

Kod korišćenja **Inline variable** opcije PSI pre i posle izgleda ovako:

Pre korišćenja

PsiMethod (printStudentDetails)

- PsiParameter (studentId)
- PsiDeclarationStatement (Student student = findStudent(studentId);)
    - PsiTypeElement (Student)
    - PsiLocalVariable (student)
        - PsiIdentifier (student)
        - PsiExpression (findStudent(studentId))
- PsiIfStatement
    - PsiExpression (student != null)
        - PsiReferenceExpression (student)
    - PsiBlockStatement
        - PsiExpressionStatement (printStudentInfo();)
    - PsiElseStatement

        - PsiBlockStatement
            - PsiExpressionStatement (System.out.println("Student not found");)

Posle korišćenja

- PsiMethod (printStudentDetails)
    - PsiParameter (studentId)
    - PsiIfStatement
        - PsiExpression (findStudent(studentId) != null)
            - PsiMethodCallExpression (findStudent(studentId))
        - PsiBlockStatement
            - PsiExpressionStatement (printStudentInfo();)
        - PsiElseStatement
            - PsiBlockStatement
                - PsiExpressionStatement (System.out.println("Student not found");)

Razlike u PSI Strukturi kod Inline

Pre Refaktorisanja

- Postoji PsiDeclarationStatement za promenljivu student:
    - PsiTypeElement za tip Student
    - PsiLocalVariable za promenljivu student
    - PsiExpression za dodelu vrednosti promenljivoj student

Nakon Refaktorisanja

- PsiDeclarationStatement za promenljivu student je uklonjen.
- PsiExpression u PsiIfStatement direktno koristi rezultat poziva metode findStudent(studentId).

Ubačeni su samo ključni delovi iz PSI.