Programok statikus ellenőrzése

Önálló laboratórium feladat összefoglalója

Fejes Endre (G2MMW3)
Konzulens: Majzik István
BME Méréstechnika és Információs Rendszerek Tanszék
Rendszertervezés ágazat, 2008/2009. II. félév

A statikus hibaellenőrzés a programok kódjának elemzését jelenti. Ezzel a módszerrel nem deríthető fel minden probléma és a megtaláltak se minden esetben jelentenek tényleges hibaforrást. Az ellenőrzés során bizonyos forráskód mintákat keresünk, amik valószínűleg hibára utalnak.

Külső konzulenssel egyeztetve a konkrét feladat Java forrásfájlokban nem használt elemek – osztályok, metódusok – felderítése, és a kódból kitörlése lett. Először kutatómunkát végeztem, hogy felderítsem, nincsenek-e erre a feladatra már kész, felhasználható eszközök. Meglepetésül szolgált, hogy erre a konkrét, viszonylag egyszerű feladatra nincs még megoldás. Két lehetőség adódott. Egyik, hogy egy más funkciókat megvalósító, de bővítményeket is kezelő eszközhöz fejlesszek kiegészítést, másik, hogy egy különálló programot készítsek, valamilyen fejlesztőkörnyezetbe integrálható módon. Mivel utóbbihoz a kiindulási alap már megvolt a külső cégnél, ezért végül ezt, azaz egy *Eclipse plugin* fejlesztését választottam.

A feladat három fő lépésből állt. Elsőként a Java forrásfájlokat kellett feldolgozni, a relációkat elemezni. Ezután az előre megadott szükséges elemekhez (továbbiakban belépési pontok) hozzá kellett venni a hivatkozottakat. Végül egy jelentést készíteni, ami alapján kiválaszthatók a szükségtelen elemek, majd ezeket eltávolítani a forráskódból.

Mivel a program egy *Eclipse* bővítmény, így evidens, hogy forrásfájlok bemeneteként egy project szolgál, benne kódfájlokkal, csomagokkal.

Az első lépést *Abstract Syntax Tree* (AST) technológiával oldottam meg, amihez az *Eclipse* nyújtotta Java *Deveplopment Tools* (JDT) eszközt használtam. Az JDT-beli AST nem más, mint a Java nyelv meta-modelljét leíró szerkezet, aminek elemeinek használatával egy Java kódot tudunk modellezni. Amikor az AST meta-modellből létrehozunk egy példányt akkor már *Concrete Syntax Tree*-ről (CST) beszélünk. A JDT által biztosított CST szerkezetet lemásoltam a könnyebb bővíthetőség érdekében (az eredeti struktúra könnyen bejárható a *Visitor* osztályokkal, de a funkciók köre nehezen bővíthető).

A hasznos elemeket egy belépési pont listában adhatjuk meg, *Fully Qualified Domain Name* (FQDN) formátumban, azaz teljes névvel. Ezek lesznek a CST bejárásánál a kiindulási pontok. Ezután az AST által leírt hívási gráfot járjuk be mélységi bejárással, és minden érintett elemet megjelölünk. Ilyen módon a kiindulási pontokkal egy izolált részgráfban lévő elemek lesznek megjelölve, amik pontosan a szükségesek.

A jelölések során egy olyan problémába ütköztem, hogy néha előfordul olyan metódus, amit implementálunk, mert szükség van rá (*Thread.run()*), azonban a Java osztályokon keresztül egy másik metódussal hívjuk meg (*Thread.start()*). Ezt, elkerülve a szükséges Java könyvtárak nagy költségű elemzését, álnév listával oldottam meg, ahol megadhatjuk, hogy egy osztályon (vagy leszármazottjain belül) melyik metódusok hivatkoznak egymásra.

A bejárás után jelentést generáltam az elemek és hivatkozásaik illetve jelölésük listázásával és ezt egyrészt egy fájlba mentem, másrészt egy *Eclipse* nézetbe fa struktúrába töltöm be, ahol a jelölések felülbírálhatóak és kiadható a tisztítás parancs. Ez még további fejlesztést igényel, mert például az importok nincsenek kezelve.