Facultatea de Automatică și Calculatoare
Programul de Licență: Calculatoare și Tehnologia Informației

XCore: Suport pentru dezvoltarea de instrumente de analiză a programelor

Rezumat Proiect Dimplomă

Alexandru Ștefănică

Coordonator
Dr. Ing. Petru-Florin Mihancea

Timişoara Iunie, 2015

2

> Any fool can write code that a computer can understand. Good programmers write code that humans can understand
>
> Martin Fowler

Pentru orice sistem software vrem să fim în stare să verificăm/să evaluăm:

- calitatea sistemului, a codului scris, a design-ului
- test coverage
- respectarea standardelor impuse
- folosirea sau nu a hack-urilor de limbaj.

Pentru a putea fi în stare să facem acest lucru avem nevoie de tool-uri pentru analiză de programare. Desigur, există nenumărate programe care fac acest lucru:

Wala IBM analiză de fluxului de data, analiză cu respect la context, . . .

FindBugs depistează erorile din codul sursă

IntelliJ O platformă care implementează 100+ de analizatori pentru cod.

SonarJ Monitorizarea performanței aplicației

CodePro Platformă care integrează numeroase metrici pentru software analysis. Implementat de LOOSE Group.

Toate aceste tool-uri nu sunt create să ruleze doar pe un anumit cod, ci pe o clasa întreagă de programe. Pentru acest lucru ele definesc un metamodel. Un metamodel reprezintă elementele necesare descrieri unui model, e.g în cazul codului sursă metamodelul reprezintă gramatica limbajului. În figura 1 se poate observa că există trei clase A, B, C care sunt descrise fară probleme de metamodelul Class. Tot în această figură se poate observa că metamodelul Class nu este singurul care poate descrie clasele A, C, ci și metamodelul MyClassEntity poate face acest lucru. Însă cum

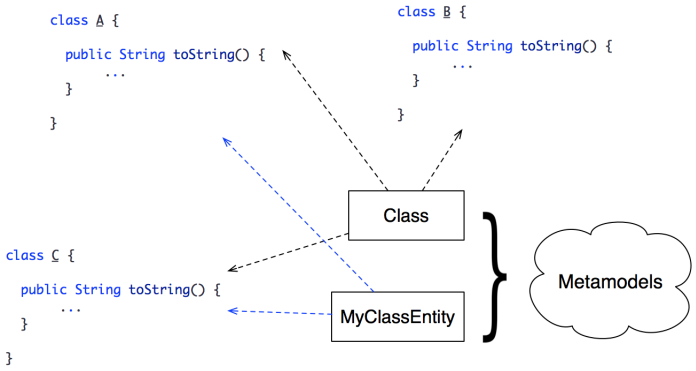

Figura 1: Două metamodele diferite pentru același model

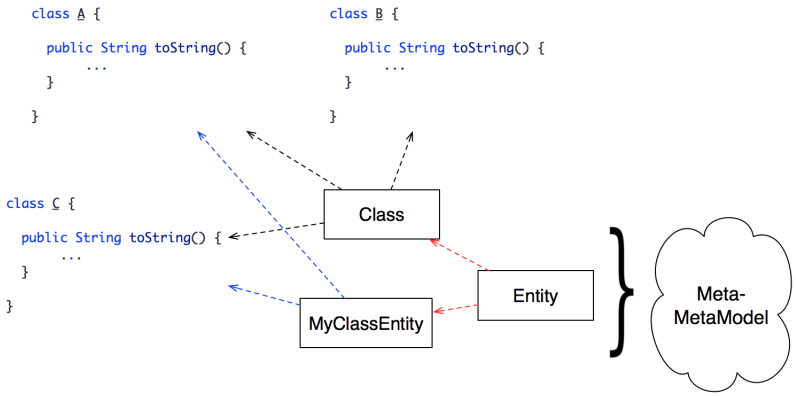

Figura 2: Meta-Metamodele

am putea să le unim ? Cum am putea să descriem aceste metamodele ? Pentru a putea face toate aceste lucruri într-un mod uniform avem nevoie de un meta-metamodel. Un exemplu de astfel de meta-metamodel este prezentat și în figura 2.

Aceeași problemă apare și în cazul tool-urilor. Fiecare tool vine cu propriul său metamodel care trebuie integrat în platformă. Noi ne dorim să creăm o platformă care este ușor extensibilă și care permite integrarea cu ușurință a mai multor tool-uri, plugin-uri. Pentru acest lucru, însă, avem nevoie de un meta-metamodel.

CodePro are un meta-metamodel care este ușor extensibil. Este foar ușor de adăugat o nou metrică: trebuie extinsă clasa PropertyComputer așa cum face și clasa CycloAVG în figura 3.

Tot în acestă figură se pot observa marcate cu roșu anumite

```
public class CycloAVG extends  PropertyComputer {
    public CycloAVG(String name, String longName, String entityType,
            String resultEntityTypeName) {

super("CycloAVG2", "Average cyclomatic complexity", "class", "numerical");
    }

    public ResultEntity compute(AbstractEntityInterface anEntity) {
        GroupEntity methods = (GroupEntity) anEntity.contains("method
group");

        if (null == methods) {
            return new ResultEntity(0);
        }

        long sum = 0;
        long count = 0;

        for (final AbstractEntityInterface element: methods.getElements())
        {


            if (null != element.getProperty("CYCLO"))
                continue;
            }
            sum += (Integer)element.getProperty("CYCLO").getValue();
            ++count;
        }

        return new ResultEntity((1.0 * sum)/(1.0 * count) );
    }

}
```

Figura 3: CycloAVG implementat în CodePro

```
@PropertyComputer
public class CycloAVG implements
                    PropertyComputer<Double, ClassEntity> {
    public CycloAVG() {}

    public Double compute(@NotNull ClassEntity entity) {
       long sum = 0;
       int count = 0;
       for (final MethodEntity method: entity.getMethodGroup()) {

             sum += method.getCyclomaticComplexity();

             ++count;
       }

       return (1.0 * sum) / count;
    }
}
```

Figura 4: CycloAVG implementat în XCore

probleme de design ce apar datorită arhitecturii:

constante „magice” Se poate observa în constructorul clasei faptul că metrica CycloAVG se înregistrează prin numele „CycloAVG”. Însă ce se întâmplă dacă mai există o metrică cu acest nume ? Ce se întâmplă cu metricele care depind de noi dacă metrica noastră își schimbă numele ?

generalizare CycloAVG își propune să calculeze o valoare pentru clasă însă funcția compute primește ca parametru un AbstractEntity. Ce se întâmplă dacă acest entity este defapt o metodă sau pachet ?

Orice eroare datorată problemelor menționate mai sus poate fi descoperită doar la runtime. Astfel noi am reușit să devenim dynamically typed într-un limbaj statically typed. Lucru ce dăunează grav întreținerii și dezvoltării platformei.

Noi am dori să scriem codul ca în figura 4. Se poate observa clar faptul că CycloAVG procesează un ClassEntity și în același timp sunt clare toate dependințele acestei metrici. Cel mai important lucru de observat este faptul că am devenit din nou statically typed, păstrând în același timp ușurința de a extinde și integra alte elemente.

XCore permite aceste lucruri prin reimplementarea meta-metamodelului din CodePro. Meta-Metamodelul a fost implementat prin folosirea

de adnotări. Adnotări sunt elemente Java prin care se poate asocia unuei clase, metode, tip, ... etc, metainformație. În figura 4 se poate observa că @PropertyComputer adontează clasa CycloAVG. Penru a putea prelucra toate aceste metainformații avem nevoie de un procesor de adnotări. Un procesor de adnotări este un plugin pentru compilator care atașează o semantică pentru unul sau mai mutle adnotări.

Tot în figura 4 se poate observa folosirea unor interfețe pentru a asigură consitența tipurilor și pentru a specifica metamodelul necesar pentru metrica pe care o implementăm. În momentul când compilăm proiectul, procesorul de adnotări a lui XCore va fi invocat. Acesta va prelucra toate clasele adnotate și va genera codul necesar pentru metamodel metricilor implementate.

Acest tool este destul de greu de evaluat întrucât este dependent de utilizator. Pentru a îl putea evalua am reimplementat InsiderView, construind XCoreView. InsiderView este un plugin pentru eclipse dezvoltat deasupra lui CodePro și care face accesibilă platforma CodePro utilizatorilor Eclipse.

În tabelul 1 se poate observa faptul că am reușit să eliminăm complet toate constante „magice” și cast-urile care sunt responsabile, cel puțin parțial, pentru problemele menționate la început. În același timp aș dori să observați că numarul de linii de cod a rămas aproximativ același, iar complexitatea generală a codului a scăzut. Astfel putem afirma că tool-ul nostru a reușit cu succes să transforme toate plugin-urile din dynamic typed în statically typed păstrând toate proprietățiile de exentesabilitate și integrabilitate.

Tabela 1: Agregarea valorilor pentru comparare. Cele mai importante valori au fost colorate.

	Number of Lines	Cyclomatic Complexity	Number of Casts	Number of Magic Strings
Avg CodePro	34.153	4.307	1.796	2.230
Avg XCoreView	31.384	2.384	0	0
Sum CodePro	444	56	23	29
Sum XCoreView	408	31	0	0