Automatizarea procesului de detecţie a carenţelor de proiectare în sisteme orientate-obiect

Ciprian-Bogdan Chirilă
cipri_chirila@yahoo.com

18 iunie 2002

Rezumat

Datorită numărului mare de sisteme orientate-obiect monolitice şi inflexibile şi al costurilor acestora este necesară o modalitate de a le reproiecta spre a fi întreţinute sau reutilizate. Pentru a putea fi reproiectate trebuiesc eliminate carenţele de proiectare. Acest lucru este realizabil cu ajutorul strategiilor de detecţie în mod sistematic, scalabil şi repetabil.

În acest articol vom prezenta un instrument software care modelează strategiile de detecţie prin intermediul metricilor şi al operatorilor statistici, aplică aceste strategii pe sisteme moştenite şi oferă introspecţie în codul sursă exact la entitatea suspectă.

1 Introducere

În anii 80 au fost realizate un număr mare de sisteme orientate-obiect care urmau să fie dezvoltate în continuare. Prin tehnica programării orientate-obiect într-adevăr se câştigă din punct de vedere al calităţii software-ului, cât şi din punct de vedere al timpului afectat dezvoltării. În anii 90 am obţinut un număr mare de sisteme orientate-obiect pe scară largă care s-au dovedit a fi:

- *inflexibile*: nu pot fi adăugate cu uşurinţă funcţionalităţi noi
- *monolitice*: nu există o structurare a funcţionalităţii sistemului bazată pe componente

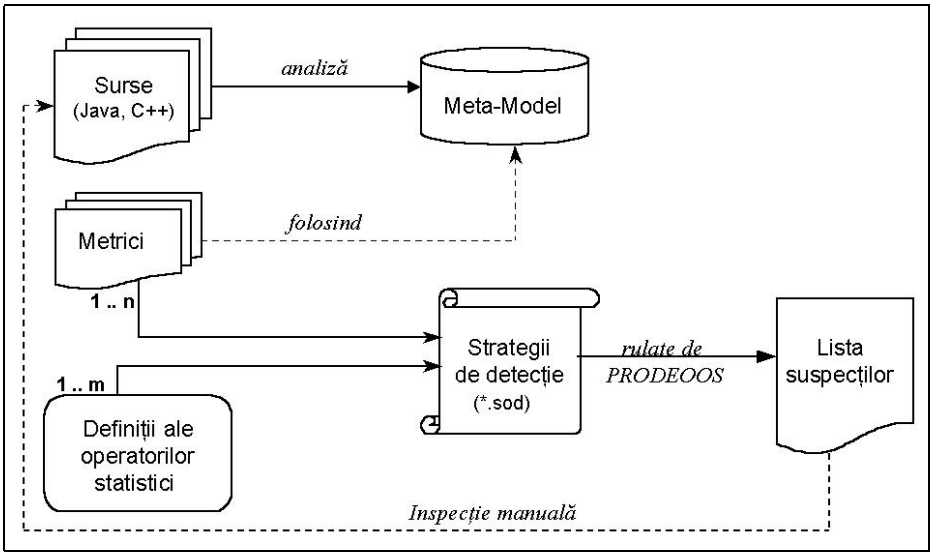

Figura 1: Abordare

- *greu de întreţinut*: încercarea de a aduce noi modificări se soldează cu un lanţ nesfârşit de ajustări în multiple locuri.

Se pune problema a ceeea ce se poate face cu un astfel de sistem software. Dacă avem de a face cu un sistem a cărui valoare economică este scăzută atunci în cazul în care el nu este flexibil se poate opta chiar pentru abandonarea lui. Problema este critică în cazul sistemelor cu valoare economică crescută. Aici se pune problema dezvoltării, chiar în condiţiile unei inflexibilităţi cronice. Aici există o necesitate imperioasă pentru *detecţia carenţelor de proiectare*, în vederea eliminării lor şi a dezvoltării ulterioare.

2 Automatizarea detecţiei

Ne propunem să automatizăm detecţia bazată pe metrici. Deşi la ora actuală detecţia se face manual şi este aplicabilă local, ne propunem să realizăm un *instrument software* care să ne ofere un proces de detecţie *general* şi *scalabil*. Abordarea se bazează pe articolul domnului Marinescu [2, Mari01a] care se referă la detecţia carenţelor de proiectare bazate pe metrici.

2.1 Metamodel

Pornind de la codul sursă al unui sistem software şi folosind instrumente software intermediare, generăm un metamodel. Acest metamodel va conţine

informaţii ce ne vor ajuta la detecţie. Informaţiile conţinute sunt referitoare la: accese la variabile, apeluri de metode, declaraţii de clase, declaraţii de variabile, relaţii de moştenire, şi sunt memorate în tabele, care sunt interogate de motorul de baze de date, ce se află sub capota instrumentului software.

2.2 Metrici

2.2.1 Definiţie

Metricile sunt funcţii care reflectă proprietăţile entităţilor software sub formă de valori numerice. Prin entităţi înţelegem: clase, metode, subsisteme.

Instrumentul software va modela metrica prin intermediul unui script SQL, asigurând un grad mare de libertate în manipularea metamodelului.

2.2.2 Exemple

Greutatea unei clase (WOC) Prin greutatea unei clase se întelege raportul dintre numărul de metode non set-get şi numărul total de membri din interfaţa clasei. Metrica măsoară gradul de funcţionalitate al clasei.

Numărul de atribute publice al unei clase (NOPA) Numele metricii este suficient de sugestiv pentru întelege ceea ce ea măsoară. Rezultatul este folositor pentru a detecta clase care depozitează date separându-le de comportamentul lor, idee contrară principiilor tehnologiei orientate-obiect.

Numărul de metode accesoare al unei clase (NOAM) Prin metode accesoare m-am referit la acele metode care au rolul de a citi sau de a scrie valoarea unui membru. Complexitatea unor astfel de metode este extrem de scăzută. Uneori programatorii ascund datele sub astfel de metode, lăsându-le singure în cadrul clasei şi astfel separându-le de comportamentul lor natural.

2.3 Operatori statistici (outlieri)

2.3.1 Definiţie

Un outlier este un operator statistic care aplicat pe colecţii de date, extrage acele date care au valori extreme. Instrumentul software va aplica operatori statistici pe colecţii de entităţi.

2.3.2 Exemple

Primele sau ultimele n valori Acest tip de outlieri după cum le spune şi numele sunt interesaţi de entităţile cu valori extreme. Este evident că lista de entităţi pe care se aplică outlier-ul este sortată. S-ar putea să ne intereseze care clase dintr-un proiect sunt cele mai "uşoare".

Valori mai mari sau mai mici decât un prag dat Un alt caz de suspiciune pot trezi entităţile care sunt mai mari sau mai mici decât o valoare dată. Entităţile surprinse cu valori sub sau peste limita legală sunt trecute pe lista suspecţilor.

2.4 Strategia de detecţie

Strategia de detecţie a unei carenţe poate fi definită în felul următor: [2] expresie cuantificabilă a unei reguli prin care entităţile afectate de carenţa respectivă să poată fi automat detectate. Pornim de la următoarea definiţie formală bazată pe metrici:

$$S := {M_1}^{O_1} * {M_2}^{O_2} * \cdots * {M_n}^{O_n}$$

$$* := \cup \quad | \quad \cap \quad | \quad \setminus$$

Pentru a opera asupra metamodelului vom avea nevoie de o structură arborescentă care să modeleze formula teoretică. De accea m-am gândit la următorul set (simplificat) de reguli:

```
DetectionStrategy     := StrategyDefinition SymbolsDefinition

# reguli pentru definirea unei strategii de detectie
StrategyDefinition    := StrategyName ":=" DetectionRule ";"
DetectionRule         := MetricWithOutliers |
                         ComposedDetectionRule
MetricWithOutliers    := "(" MetricName "," OutlierName ")"
ComposedDetectionRule := DetectionRule CompositionOperator
                         DetectionRule
StrategyName          := [A-z][A-z0-9_]
CompositionOperator   := "or" | "and" | "butnotin"

# simbolurile sunt definitii de metrici si operatori
SymbolsDefinition     := MetricDefinition |
                         OutlierDefinition
```

```
# reguli pentru definirea metricilor
MetricDefinition   := MetricName  ":=" SqlQuery ";"
MetricName         := [A-z][A-z0-9_]
SqlQuery           := [.]   # SELECT <Entity> <Value>

# reguli pentru definirea operatorilor statistici
OutlierDefinition := OutlierName ":=" OutlierType
                     "(" OutlierParameter ")" ";"
OutlierType       := "TopValues" | "BottomValues" |
                      "HigherThan"| "LowerThan" | "BoxPlots"
OutlierName       := [A-z][A-z0-9_]
OutlierParameter  := [0-9][0-9,][%]
```

Instrumentul software modelează aceste reguli printr-un analizor lexical, un analizor sintactic şi un generator de arbore. Ideea de bază a gramaticii este următoarea: o strategie poate fi o metrică simplă sau o metrică compusă dintr-un operator logic şi alte metrici. Astfel putem construi într-un mod relativ simplu strategii de detecţie de diferite complexităţi.

2.5 Exemplu de strategie de detecţie

Am ales ca exemplu o strategie simplă care să detecteze clasele de date [3, Mari01b].

```
DataClassesStrategy:=
              (
                (WOC, WOCBottom) and (WOC, WOCLower) and
                ((NOPA, NOPAOutliers) or
                (NOAM, NOAMOutliers))
              );

WOC:=
      SELECT all_pub.f_class AS f_class,
       (100*pub_mth.cnt / all_pub.cnt) AS woc FROM...
NOPA:=
      SELECT f_class, count(f_class) AS nopa
      FROM v_members...
NOAM:=
      SELECT f_class, count(F_class) AS noam
```

```
      FROM v_accessor_methods...

WOCBottom    := BottomValues(10);
WOCLower     := LowerThan(33);
NOPAOutliers := TopValues(7);
NOAMOutliers := TopValues(5);
```

Voi comenta pe scurt această strategie, care îsi propune să detecteze clasele de date. Prin clase de date înţelegem acele clase care separă datele de comportamentul lor. Bazându-ne pe această idee, construim strategia în felul următor: ne interesează acele clase care au membri mulţi şi nu prea au metode funcţionale. Pentru acestă strategie putem folosi o metrică prin care se măsoară greutatea unei clase, pentru a le detecta pe cele cu procent mic de funcţionalitate, metrica care măsoară numărul de atribute publice şi metrica care numără metodele accesor. Operatorii statistici sunt calibraţi astfel: caut clase mai uşoare de 33 procente uitându-mă la ultimele 10, caut primele 7 clase cu cele mai mari numere de atribute publice, primele 5 clase cu cele mai multe metode accesor. Am aplicat strategia "Data Class" pe un studiu de caz şi în urma detecţiei am obţinut următorul tabel:

Clasă	WOC	NOPA	NOAM
A	0.05	14	
B	0.06	16	
C	0.10		20

Un set de rezultate ca şi cel din tabelul anterior poate fi interpretat astfel: clasele A şi B sunt mai uşoare decât valoarea pragului impus şi putem conchide că ele sunt clase de date cu mulţi membri publici. În contrast cu clasele A şi B este clasa C care nu are membrii publici, dar are în schimb multe metode accesor, ascunzându-şi datele. Instrumentul software asigură introspecţia în codul sursă. Astfel se poate realiza un feedback pentru corecţia carenţelor de proiectare. Dacă facem introspecţia în codul sursă putem confirma rezultatele detecţiei.

3 Concluzii

Utilizând instrumentul software câştigăm:

- *abordare sistematică* prin descrierea strategiilor în limbajul SOD

- *repetabilitate* bazată pe faptul că strategiile de detecţie pot fi reutilizate, parametrii pot varia, dar strategie rămâne aceeaşi
- *scalabilitate* asigurată de instrumentul software - PRODEOOS cât şi de rezultatele obţinute pe sisteme industriale.

Bibliografie

[1] R. Martin. *Design Principles and Patterns.* http://www.objectmentor.com, 2000.

[2] R. Marinescu. *Detecting Design Flaws via Metrics in Object-Oriented Systems.* Proceedings of the TOOLS USA 2001, IEEE Computer Society, 2001 (to be published) .

[3] R. Marinescu. *Design Flaws and Detection Strategies.* PhD Presentation, Karlsruhe, April 2001.