

Cuprins

Introducere	1
Contribuții	6
Analiză și proiectare	7
Descrierea arhitecturii	8
Serviciul de procesare de text	9
Serviciul de generare de cod	28
Serviciul de integrare	33
Detalii de implementare	35
Detaliile serviciului de procesare de text	35
Detaliile serviciului de generare de cod	39
Detaliile serviciului de integrare	42
Exemple de utilizare	43
Direcții de viitor	49
Concluziile lucrării	50
Bibliografie	51

Introducere

O mare parte a sistemelor software din jurul nostru sunt proiectate să rezolve situații din viața de zi cu zi - o persoană are un cont bancar pe baza căruia își gestionează economiile; o bibliotecă pune la dispoziția publicului o serie de cărți, care pot fi împrumutate; un producător de autoturisme construiește diverse tipuri de mașini.

Programarea orientată obiect este o paradigmă de programare ce ajută la modelarea acestor concepte, introducând noțiunea de *obiect*. Un obiect are *attribute*, ce reprezintă informația care îl caracterizează, și *metode*, care specifică comportamentul obiectului.

Această paradigmă reduce atât diferențele semantice dintre lumea reală și un sistem software, cât și complexitatea funcțională a acestuia din urmă, datorită ușurinței și flexibilității cu care conceptele ce trebuie gestionate informatic sunt abstractizate și ajung să fie "înțelese" de către mașina care le manipulează starea.

De asemenea, G. Booch precizează în [1] că "modelul obiectului s-a dovedit a fi un concept unificator în informatică, aplicabil nu doar la limbajele de programare, ci și la proiectarea interfețelor cu utilizatorul, a bazelor de date și chiar a arhitecturilor calculatoarelor".

Deoarece situațiile din lumea reală pot fi descrise folosind limbajul natural (de exemplu, frazele din primul paragraf), considerăm utilă existența unei aplicații care **să prelucreze un text scris într-un astfel de limbaj, să extragă conceptele aliniate paradigmei obiectuale și să le modeleze în sensul descris mai sus.**

O astfel de aplicație ajută la dezvoltarea produselor software oferind un cadru de modelare rapidă a conceptelor necesare, independent de limbajul de programare. De asemenea, ea poate fi folosită și de către persoane fără

cunoștințe de programare orientată obiect, pentru familiarizare și învățare (auto-instruire).

Astfel, putem considera drept public țintă trei categorii de utilizatori:

- ❖ **Utilizatorii din mediul de afaceri**, care trebuie să comunice cerințe ale unui produs software către persoane din domeniul informatic (analiști, programatori). Aceștia cunosc modificările care trebuie aduse produsului în cauză, însă discuția cu persoanele din domeniul informatic nu este mereu o discuție ușoară. Folosind aplicația, ei pot converti cerințele lor din limbajul pe care îl cunosc (limbajul natural) în cod, pe care îl pot trimite informaticienilor.
- ❖ **Utilizatorii fără cunoștințe în domeniul informaticii**, care au nevoie de o unealtă care să arate legătura dintre obiecte și codul care modelează acele obiecte.
- ❖ **Utilizatorii dezvoltatori**, care folosesc aplicația sau consumă serviciile expuse pentru uzul în aplicațiile proprii.

În prezent, există multiple soluții de identificare a entităților și a proprietăților acestora dintr-un text scris în limbaj natural [2], [3] (concentrate, în principal, pe textele în limba engleză). De asemenea, există și servicii care, primind tipuri specifice de date de intrare, pot modela, de exemplu, diagrame [4].

O altă aplicație interesantă este Fluent Editor [5], care poate modela ontologii dintr-un limbaj natural controlat. Utilizatorul se poate folosi de expresii predefinite precum "every", "no" sau "is-a" pentru a exprima proprietăți și relații între entități. Rezultatul este o ontologie care reprezintă aceste relații și poate răspunde la întrebări de tipul "Who is-a?". Un exemplu de utilizare poate fi regăsit în [6].

Aceste tipuri de aplicații, fie prin limbaj natural sau prin date de intrare într-un format specific, pot modela concepte din viața reală, însă rezultatul nu este transpus natural în cod.

În schimb, există și diverse medii interactive de învățat programare și generare de cod. Blockly [7] este o aplicație dezvoltată de Google care reprezintă conceptele de programare ca blocuri grafice ce pot fi interconectate. Utilizatorul așează blocurile și formează un program logic, iar aplicația generează codul rezultat în diverse limbaje de programare.

Însă pentru a folosi Blockly și alte aplicații asemănătoare, utilizatorul trebuie să învețe și să se obișnuiască cu metodele proprii de reprezentare a conceptelor de programare.

Obiectivul acestei lucrări este prezentarea unei soluții de modelare care să răspundă tuturor acestor condiții: transformarea unui text scris în limbaj natural în cod orientat obiect, adică extragerea și modelarea conceptelor din text și generarea codului cu un efort minim din partea utilizatorului.

Soluția aleasă se bazează pe multiple *servicii web*: un serviciu de prelucrare a textului natural, care extrage tipurile de concepte folosind o *gramatică* și construiește o structură de modelare, un serviciu de generare de cod care, plecând de la structura construită anterior, generează clase și metode în multiple limbaje de programare și un serviciu de integrare, care gestionează tot procesul.

Am ales această abordare pentru a crește flexibilitatea utilizării: un utilizator poate fi interesat doar de un singur pas din cei expuși (de exemplu, un dezvoltator poate să-și creeze propriile servicii consumatoare pentru serviciul de generare).

Astfel, în secțiunea "Contribuții" vom prezenta succint principalele noastre realizări; în capitolul "Analiză și proiectare" vom descrie pe larg cerințele aplicației, urmând ca în capitolele "Descrierea arhitecturii" și "Detalii de implementare" să prezentăm în detaliu structura serviciilor, algoritmii și bibliotecile folosite.

În capitolul "Exemple de utilizare" se regăsesc multiple exemple de apelare și consumare a serviciilor expuse, iar în ultimul capitol, "Direcții de viitor", vom prezenta câteva posibile extensii ale aplicației.

1. Contribuții

Principalele contribuții personale sunt:

- ❖ Crearea unei gramatici expresive care validează textul scris în limbaj natural și creează arbori propoziționali.
- ❖ Construcția unor arbori propoziționali care să faciliteze modelarea conceptelor orientate obiect.
- ❖ Propunerea "modelului-dicționar", o structura de date gândită pentru exprimarea conceptelor de orientare obiect, independent de un limbaj de programare.
- ❖ Crearea unui generator de cod ușor extensibil care, primind un model-dicționar, analizează validitatea acestuia și generează cod orientat obiect în multiple limbaje de programare.
- ❖ Agregarea acestor module și funcționalități în trei servicii web.

2. Analiză și proiectare

Se dorește o aplicație care, la primirea unui text scris într-un limbaj natural controlat, în limba engleză, și a unui limbaj de programare, să extragă din text modelul ce poate fi reprezentat cu ajutorul conceptelor din paradigma de programare orientată obiect și să genereze codul ce este asociat modelului, în limbajul de programare specificat.

Concret, trebuie să putem extrage și modela noțiunile de:

- ❖ clasă,
- ❖ moștenire,
- ❖ clasă abstractă,
- ❖ interfață,
- ❖ clasă finală,
- ❖ membru,
- ❖ metodă,
- ❖ constructor,
- ❖ modificador de acces.

De asemenea, aplicația trebuie să fie extensibilă relativ la:

- ❖ conceptele pe care le modelează (să fie posibilă și facilă adăugarea unui concept nou sau ștergerea unuia vechi),
- ❖ modalitatea de căutare a conceptelor în textul primit, fiind posibilă o multitudine de asocieri între un concept și textul natural care îl exprimă,
- ❖ limbajul de programare în care se generează codul rezultat.

3. Descrierea arhitecturii

Soluția propusă atinge cerințele menționate într-o manieră bazată pe servicii web[10]. Astfel, am creat o aplicație compusă din două servicii independente (**serviciul de procesare de text** și **serviciul de generare de cod**) și un **serviciu de integrare** ce concatenează fluxul celorlalte două servicii. Concatenarea se realizează cu ajutorul unei **scheme de modelare independentă de limbaj**. Apelând serviciul de integrare, din textul primit se obține codul modelator.

Am ales serviciile web deoarece oferă o modalitate facilă în ideea extensibilității. Dacă o anumită componentă a aplicației trebuie înlocuită complet, ea poate fi implementată independent de celelalte existente sau de versiunea ei anterioară. De asemenea, folosind servicii web avem acces la utilizarea codurilor de stare HTTP, o modalitate expresivă de a marca evenimente în cadrul fluxului aplicației.

În continuare, vom descrie planul arhitectural a celor trei servicii menționate mai sus.

3.1. Serviciul de procesare de text

Acest serviciu se ocupă cu identificarea în textul de intrare a conceptelor cerute și expunerea acestora într-o manieră ce nu depinde de niciun limbaj de programare.

Modalitatea prin care se execută acest flux are la bază o gramatică[12]. Aceasta pune restricțiile textului pentru a garanta existența datelor utile. Această gramatică este bazată pe identificarea anumitor combinații de cuvinte cheie pentru a identifica un anumit concept.

Spre exemplu sintagma "is a", reprezintă relația de moștenire, iar sintagma "has a" definește existența unei variabile în cadrul unei clase. Tot cu ajutorul acestei gramatici, putem crea arborii propoziționali care pot fi prelucrați pentru a culege informațiile necesare modelării. În sfârșit, datele culese sunt trimise sub o formă de dicționar ce modelează conceptele cerute.

Deoarece structura este sub formă de dicționar, aceasta nu este dependentă de niciun limbaj de programare. Acesta este primul pas în satisfacerea extensibilității relativ la limbajul în care va fi generat codul final.

În diagrama următoare, prezentăm pașii prin care textul primit la intrare este procesat; apoi vom detalia pașii cei mai importanți din punctul de vedere al aportului adus serviciului.

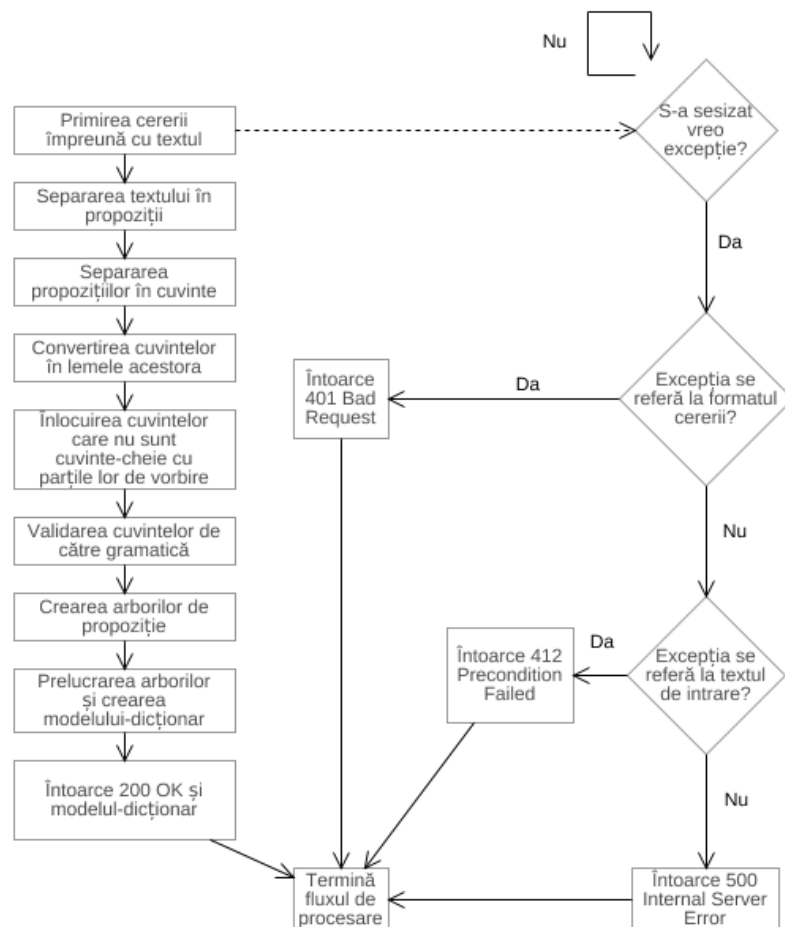


Figura 3.1. Diagrama fluxului serviciului de procesare

În prima parte a fluxului serviciului, până la validarea cuvintelor de către gramatică, regăsim pași de prelucrare care fie elimină din text informația redundantă pentru modelare, fie aduc textul într-o formă ce facilitează procesarea. Aceștia vor fi detaliați explicit în capitolul 4.1.

Validarea cuvintelor este făcută în funcție de gramatică. Gramatica aleasă este bazată pe *cuvinte cheie* și *părți de vorbire*, astfel:

- ❖ Fiecare concept referitor la proprietățile claselor, obiectelor, variabilelor, metodelor sau parametrilor unei metode corespunde unui cuvânt cheie sau unei combinații de cuvinte cheie. Astfel, textul de la intrare trebuie să fie structural controlat, să satisfacă anumite șabloane de cuvinte cheie.
- ❖ Fiecare nume de clasă, variabilă, metodă sau parametru al unei metode este asociat cu o parte de vorbire în limba engleză. Astfel, textul trebuie să aparțină unui vocabular controlat din punctul de vedere al regulilor sintactice. Spre exemplu, numele unei metode ar trebui să fie un verb, sau numele unei variabile să fie substantiv.

În cele ce urmează, vom descrie gramatica propriu-zisă.

Aceasta a fost concepută ținând cont de următoarele priorități:

- ❖ tratarea fiecărei propoziții individual,
- ❖ posibilitatea de a atinge cerințele de modelare (expresivitatea),
- ❖ calitatea textului din punctul de vedere al limbii engleze (naturaletă).

Pentru a atinge primul obiectiv, gramatica este construită astfel încât să analizeze doar o propoziție; pentru a procesa un text întreg, este necesară separarea acestuia în propoziții, care sunt prelucrate una câte una.

Pentru a atinge obiectivul de expresivitate, am separat fiecare propoziție în 4 tipuri:

```
sentence      ->  descriptive_sentence  
                |  possessive_sentence  
                |  behavioral_sentence  
                |  singleton_sentence
```

- ❖ Propozițiile "*descriptive*" se referă la modelarea conceptelor legate strict de clase, spre exemplu proprietatea unei clase de a avea un modificador de acces sau de a putea fi clasă finală.
- ❖ Propozițiile "*possessive*" se referă la modelarea conceptelor legate de variabilele unei clase și sunt la rândul lor împărțite în 3 categorii:

```
possessive_sentence  ->  defining_posessed  
                        |  posessed_limitation  
                        |  posessed_description
```

- Propozițiile "*defining_posessed*" au ca țintă propoziții ce definesc o variabilă a unei clase.
 - Propozițiile "*posessed_limitation*" au rolul de a schimba modificadorul de acces al unei variabile.
 - Propozițiile "*posessed_description*" asociază unei variabile tipul acesteia sau proprietățile de a fi constantă sau statică.
- ❖ Propozițiile "*behavioral*" modelează comportamentul unei clase, adică metodele sale. Acestea definesc numele metodei, modificadorul de acces al acesteia și o înșiruire de parametri.

- ❖ Propozițiile "*singleton*" modelează o clasă care respectă șablonul de proiectare "Singleton"[11].

Naturaletăea propozițiilor prelucrate de gramatică vine din alegerea cuvintelor cheie și a părților de vorbire. Alegerea a fost făcută astfel încât propozițiile prelucrate să fie corecte în limba engleză și înțelesul semantic al propozițiilor să fie același cu înțelesul conceptelor orientate obiect ce vor fi modelate.

Pentru a pune în prin plan acest aspect (naturaletăea), în tabelul următor vom prezenta conceptele modelate și sintagmele ce le sunt asociate.

Conceptul modelat	Sintagma asociată
Moștenire	C1 is a C2
Clasă abstractă	C1 is abstract
Interfață	C2 is an interface
Clasa finală	C3 has no heir
Variabilă	C3 has a V1 sau C3 has V1
Variabilă de tip colecție	C3 has many V2 s
Tipul unei variabile	C3' s V1 is a T
Variabilă constantă	C3' s V1 is unchangeable
Variabilă statică	C3' s V1 is shareable

Verb ce exprimă comportament (behavioral_verb)	can sau do sau should sau may
Metodă	C3 behavioral_verb M1
Metodă cu un parametru	C3 behavioral_verb M2 with P1 sau C3 behavioral_verb M2 with a P1
Metodă cu un parametru de tip colecție	C3 behavioral_verb M2 with many P1s
Metodă cu mulți parametri	C3 behavioral_verb M3 with a P1 and P2 and many P3s
Modificatorul de acces "public"	everyone knows
Modificatorul de acces "protected"	not everyone knows
Modificatorul de acces "private"	no one knows sau only
Singleton	there is only one C4

Tabel 3.1. Asocierea între concepte și sintagme

În tabel se întâlnesc următoarele prescurtări:

- ❖ C.număr reprezintă numele unei clase.
- ❖ V.număr reprezintă numele unei variabile.
- ❖ M.număr reprezintă numele unei metode.
- ❖ P.număr reprezintă numele unui parametru.

"," reprezintă operația uzuală de concatenare a șirurilor de caractere.

Aceste nume sunt validate în gramatică relativ la partea lor de vorbire din limba engleză.

Mai jos este definită gramatica integral, cu toate asocierile între concepte și combinații de cuvinte cheie și părți de vorbire.

```
sentence -> descriptive_sentence
          | possessive_sentence
          | behavioral_sentence
          | singleton_sentence
descriptive_sentence -> protected_entity
                        | entity_sealed
                        | entity_abstract
                        | entity_interface
                        | entity_desc_verb
                        | numeral_indicator
                        | inherited_entity
inherited_entity -> entity
possessive_sentence -> defining_posessed
                        | posessed_limitation
                        | posessed_description

defining_posessed -> entity_pos_verb
                       posessed
                       | private_indicator
                       entity
                       pos_verb
                       posessed
```

```

possessed_limitation -> limited_indicator
                        entity pos_indicator
                        possessed
limited_indicator      -> private
                        | protected
possessed_description -> entity pos_indicator
                        possessed desc_verb
                        description_type
description_type      -> entity
                        | numeral_indicator entity
                        | constant
                        | static
possessed             -> pos_object
                        | numeral_indicator
                        pos_object
behavioral            -> entity behav_verb
                        behavior
                        | private_indicator entity
                        behav_verb behavior
behavior              -> behav_object
                        | behav_object
                        involved_indicator
                        parameter_enumeration
parameter_enumeration -> parameter
                        | parameter
                        enumeration_indicator
                        parameter_enumeration
parameter            -> entity
                        | numeral_indicator entity
singleton            -> there_keyword desc_verb
                        private_indicator one_keyword
                        entity

```



```

protected -> negation_indicator
              everyone_keyword ack_verb
private      -> negation_indicator one_keyword
              ack_verb
sealed -> pos_verb negation_indicator
              heir_keyword
abstract -> desc_verb abstract_keyword
interface -> desc_verb interface_keyword
constant -> "unchangeable"
static -> "shareable"
private_indicator -> "only"
pos_indicator -> "POS"
involved_indicator -> "with"
enumeration_indicator -> "and"
numeral_indicator -> "a" | "an" | "many"
negation_indicator -> "no" | "not"
desc_verb -> "be"
pos_verb -> "have"
behav_verb -> "can" | "should" | "do" | "may"
ack_verb -> "know"
entity      -> "NN" | "NNP" | "JJ" | "VB" |
              "VBP" | "FW" | "DT" | "PRP" | "RB"
pos_object -> "NN" | "NNP" | "JJ" | "VB" |
              "VBP" | "FW" | "DT" | "PRP" | "RB"
behav_object -> "DT" | "RB" | "VBP" | "VB"
everyone_keyword -> "everyone"
one_keyword -> "one"
abstract_keyword -> "abstract"
interface_keyword -> "interface"
heir_keyword -> "heir"
there_keyword -> "there"

```

Prescurtările din gramatică simbolizează părțile de vorbire utilizate (din limba engleză). Acestea sunt explicate la [14].

Abordarea problemei cu această gramatică facilitează atât extensibilitatea relativă la conceptele ce trebuie modelate, adăugarea și ștergerea conceptelor realizându-se firesc, cât și la modalitatea căutării conceptelor, deoarece asocierea conceptelor cu construcțiile de text este intuitivă.

În continuarea fluxului, prin prelucrarea textului cu ajutorul gramaticii, se creează arborii propoziționali. Aceștia conțin datele de modelare pentru fiecare propoziție. În esență, arborii propoziționali sunt doar arborii creați de analizatorul gramatical (parser) al gramaticii descrise. Pentru a le analiza structura, vom alege trei exemple de propoziții și vom arăta conținutul arborilor.

Exemplul 1 :

"Bob is a human."

Exemplul 2:

"Bob has many boats."

Exemplul 3:

"Bob can sail with a boat and many friends."

Arborii creați pe baza acestor exemple se găsesc ilustrați la figurile 3.2, 3.3 și 3.4, respectiv. Cele două detalii importante la construcția acestor arbori sunt:

- ❖ Fraza de la intrare se găsește dacă concatenăm toate frunzele arborelui.
- ❖ Fiecare frunză ce reprezintă un concept modelabil (o clasă, o variabilă, o metodă sau un parametru) este descrisă cu ajutorul parcurgerii arborelui de la rădăcină spre aceasta.

Datorită primului detaliu, cuvintele din frază pot fi înlocuite cu părțile acestora de vorbire înainte de prelucrarea analizatorului gramatical și înlocuite în arbore după validare.

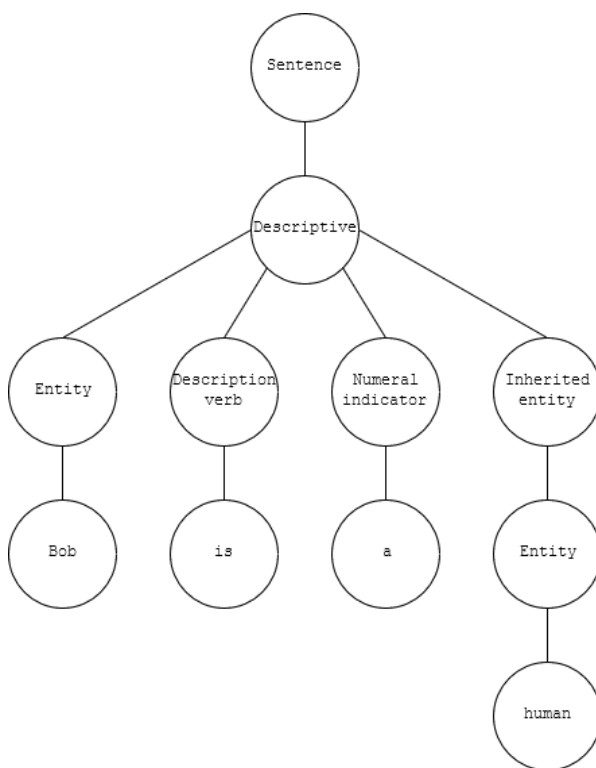


Figura 3.2 Arborele pentru exemplul 1

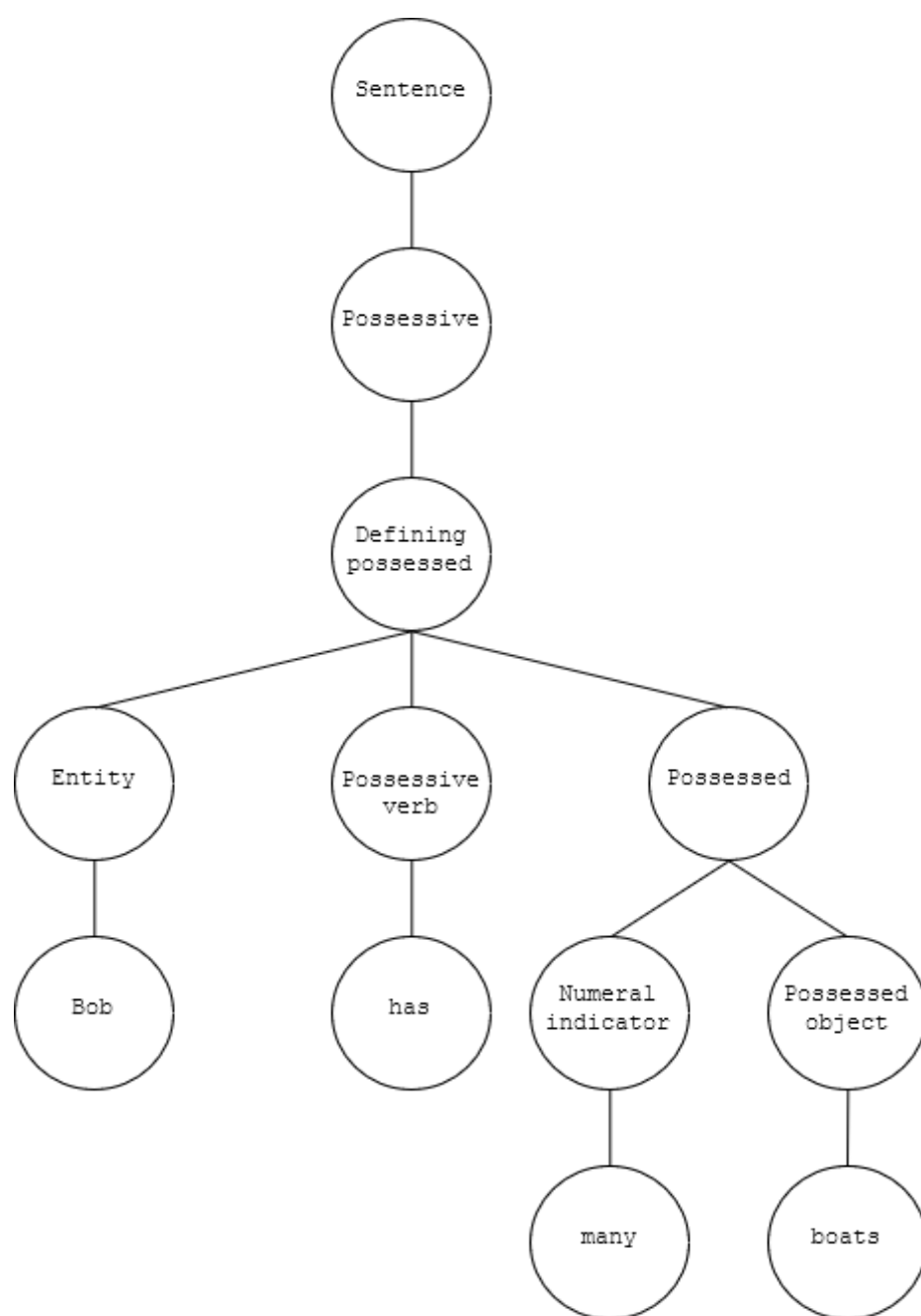


Figura 3.3. Arborele pentru Exemplul 2

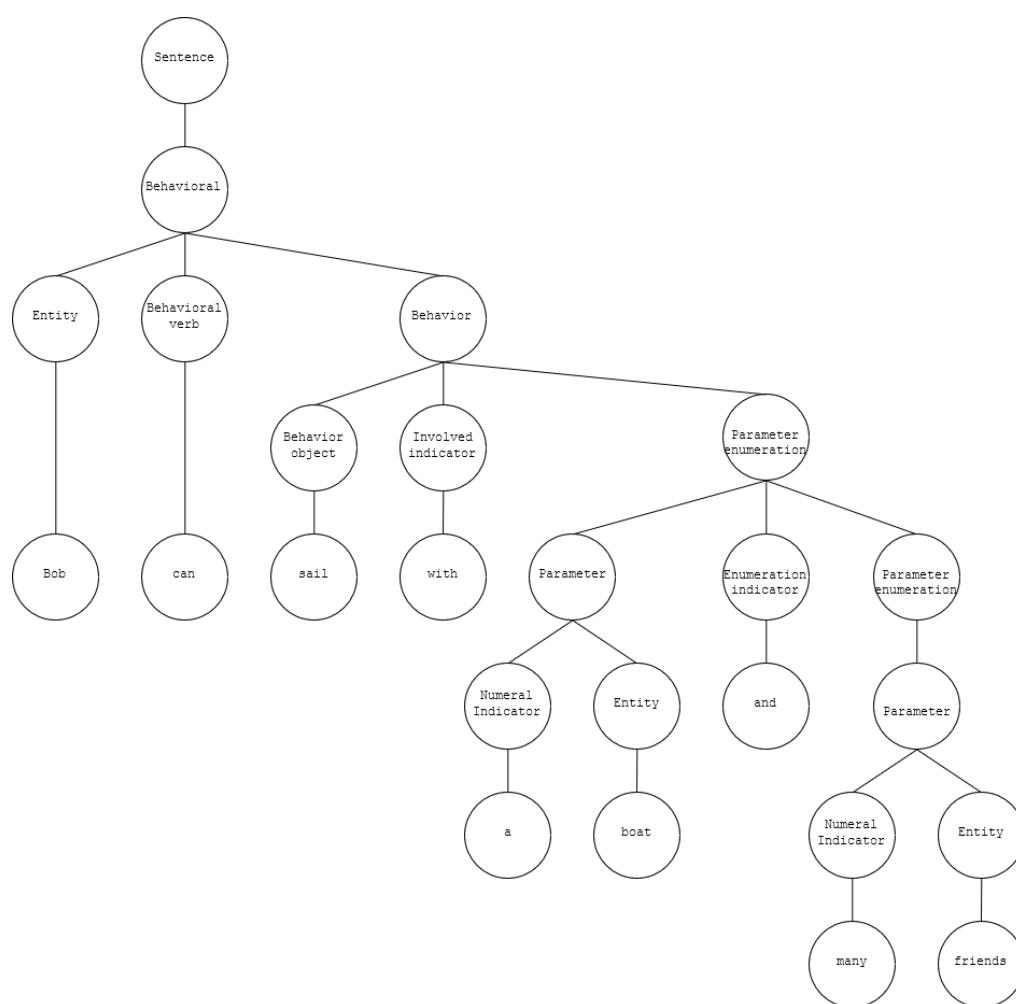


Figura 3.4. Arborele pentru exemplul 3

Arborii sunt parcurși ținând cont de cele două aspecte menționate, pentru a se realiza extragerea datelor. Detalii despre algoritmul de parcurgere există în capitolul 4.1.

Pentru satisfacerea cerinței de extensibilitate relativă la limbajul de programare în care este generat codul final, datele din arbore trebuie ținute într-o formă independentă de un limbaj de programare.

Abordarea concepută pentru această cerință este bazată pe *independența de limbaj* a structurilor de date de tip liste sau dicționare. Astfel, am creat o structură de date numită **model-dicționar**. La aceasta ne vom referi și ca schema-dicționar sau ca text procesat.

Modelul-dicționar este o structură de date concepută cu ajutorul listelor și a dicționarelor. Astfel, un model-dicționar este o listă de structuri de tip *entitate*. O *entitate* este o structură de date de tip cheie-valoare, unde cheile sunt informațiile modelabile la o clasă. În valorile cheilor unei entități se găsesc: o listă de *variabile*, o listă de *constructori* și o listă de *metode*; o variabilă este o structură de date de tip cheie-valoare ce are sarcina modelării variabilelor unei clase, un constructor este o structură de date de tip cheie-valoare ce modelează proprietățile unui constructor, iar o metodă este o structură de date de tip cheie-valoare ce modelează proprietățile unei metode. În valorile cheilor unei metode (sau unui constructor) se găsește o listă de parametri, o structură de tip cheie-valoare responsabilă cu modelarea parametrilor unei metode (sau unui constructor).

O variantă de modelare mai puțin expresivă se regăsește în [8], care a servit ca sursă de inspirație pentru modelul-dicționar.

Modelul-dicționar interpretează conceptele ce trebuie modelate de către aplicație în felul următor:

- ❖ Clasele și interfețele sunt structuri de tip dicționar care au drept chei proprietățile lor modelabile (modificatorii de acces, moștenirea etc.) și drept valori semnificația dorită ("public" pentru modificatorul de acces, de exemplu). Ne vom referi la ele ca și entități. Acestea au la rândul lor liste de membri - o listă de variabile, o listă de metode și o listă de constructori.
- ❖ Variabilele sunt structuri asemănătoare entităților din punct de vedere arhitectural. Diferența apare strict din punctul de vedere al conceptelor posibile ce definesc o variabilă.
- ❖ De asemenea, și metodele și constructorii seamănă arhitectural și diferă conceptual cu entitățile. În plus, acestea au o listă de parametri.
- ❖ Parametrii sunt modelați doar din punctul de vedere al tipului acestora.

Având această structură, modelul-dicționar este definit printr-o listă de entități.

Toate conceptele modelabile de modelul-dicționar se găsesc în diagrama de clase descrisă mai jos. De asemenea, valorile posibile ale șirurilor de caractere ce expun concepte vor fi detaliate în cadrul capitolului 4.2.

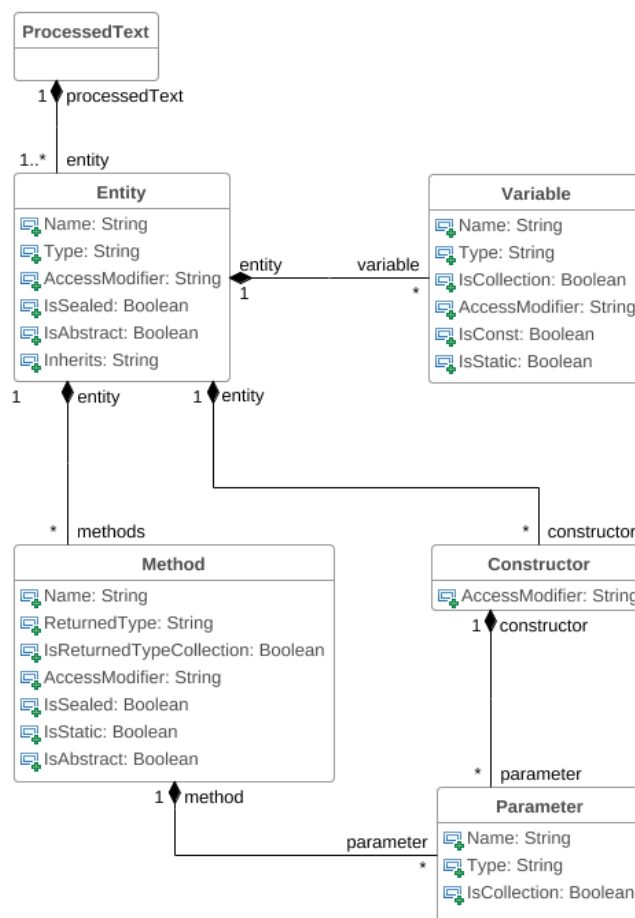


Figura 3.5. Diagrama modelului-dicționar

Această structură este rezultatul procesării de text din cadrul serviciului prezentat. Mai multe detalii despre aceasta se vor furniza și la descrierea serviciului de generare de cod, care, primind-o la intrare, va genera codul așteptat.

În finalul descrierii acestui serviciu vom prezenta un exemplu pentru fluxul acestui modul, spre mai buna înțelegere a acestuia. Astfel, vom lua un text de intrare și vom expune fiecare transformare din decursul fluxului.

Fie textul:

"Bob has a car. Bob's car is a Toyota. Bob can drive."

După pasul de primire a inputului, următorii pași sunt separarea în propoziții, apoi în cuvinte. Astfel, textul de la intrare devine o listă de propoziții, pe urmă o listă de liste de cuvinte.

```
[[ 'Bob', 'has', 'a', 'car'],  
[ 'Bob', "'s", 'car', 'is', 'a', 'Toyota'],  
[ 'Bob', 'can', 'drive']]
```

Următorul pas este aducerea fiecărui cuvânt la forma din dicționar sau la "lemma" (eng) acestuia. Acest pas ajută la simplificarea complexității unei propoziții, făcând prelucrarea mai ușoară. După acest pas, exemplul nostru va arăta astfel:

```
[[ 'bob', 'have', 'a', 'car'],  
[ 'bob', "'s", 'car', 'be', 'a', 'toyota'],  
[ 'bob', 'can', 'drive']]
```

Pasul ce succede aducerea cuvintelor la forma din dicționar este cel de identificare a cuvintelor cheie și de înlocuirea celorlalte cuvinte cu partea lor de vorbire din limba engleză.

```
[[ 'NN', 'have', 'a', 'NN'],
[ 'NN', 'POS', 'NN', 'be', 'a', 'NN'],
[ 'NN', 'can', 'VB']]
```

În continuare, fiecare listă este validată și prelucrată de analizatorul gramatical. Acesta va genera arborii propozițional.

```
(sentence
  (possessive
    (defining_possestsed
      (entity NN)
      (pos_verb have)
      (possestsed (numeral_indicator a) (pos_object NN))))
(sentence
  (possessive
    (possestsed_description
      (entity NN)
      (pos_indicator POS)
      (possestsed (pos_object NN))
      (desc_verb be)
      (description_type (numeral_indicator a) (entity NN))))
(sentence
  (behavioral
    (entity NN)
    (behav_verb can)
    (behavior (behav_object VB))))
..
```

Figura 3.6. Exemplul de arbore creat în cursul procesării

Datorită proprietății arborilor de a obține propozițiile concatenând frunzele lor, frazele se reconstituie cu cuvintele originale. Datele din arbori sunt analizate și transformate în text procesat.

```
[{
  "name": "Bob",
  "methods": [{
    "accessModifier": "public",
    "name": "drive"
  }],
  "variables": [{
    "isCollection": false,
    "name": "car",
    "type": "Toyota",
  }]
},
{
  "name": "Toyota"
}]
```

Structura de mai sus se întoarce către solicitantul cererii. O putem folosi pentru a modela diverse obiecte, sau o putem trimite mai departe la serviciul de generare de cod, pentru a o transforma într-un model specific unui limbaj de programare.

3.2. Serviciul de generare de cod

Acest serviciu primește un model structurat sub forma descrisă la Figura 3.5 și un limbaj de programare. Primul detaliu analizat este compatibilitatea cu limbajul de programare; apoi, se verifică validitatea modelul primit, garantând ieșirea unui cod compilabil (în cele mai multe dintre cazuri) în urma interpretării modelului.

Criteriile de validare sunt împărțite în două categorii: criterii sintactice și criterii semantice. Procesul de validare din punctul de vedere al acestor două criterii se execută independent. În urma validării, modelul este analizat de un generator specific limbajului de programare dat la intrare ce convertește modelul-dicționar în cod compilabil.

În Figura 3.7 este descris fluxul complet în cadrul serviciului.

Cum am menționat, primul pas în fluxul serviciului este cel de validare a limbajului de programare. Validarea este făcută prin verificarea dacă există un generator pentru limbajul respectiv. În cazul în care limbajul de programare dat la intrare este compatibil cu aplicația, se continuă fluxul în serviciu.

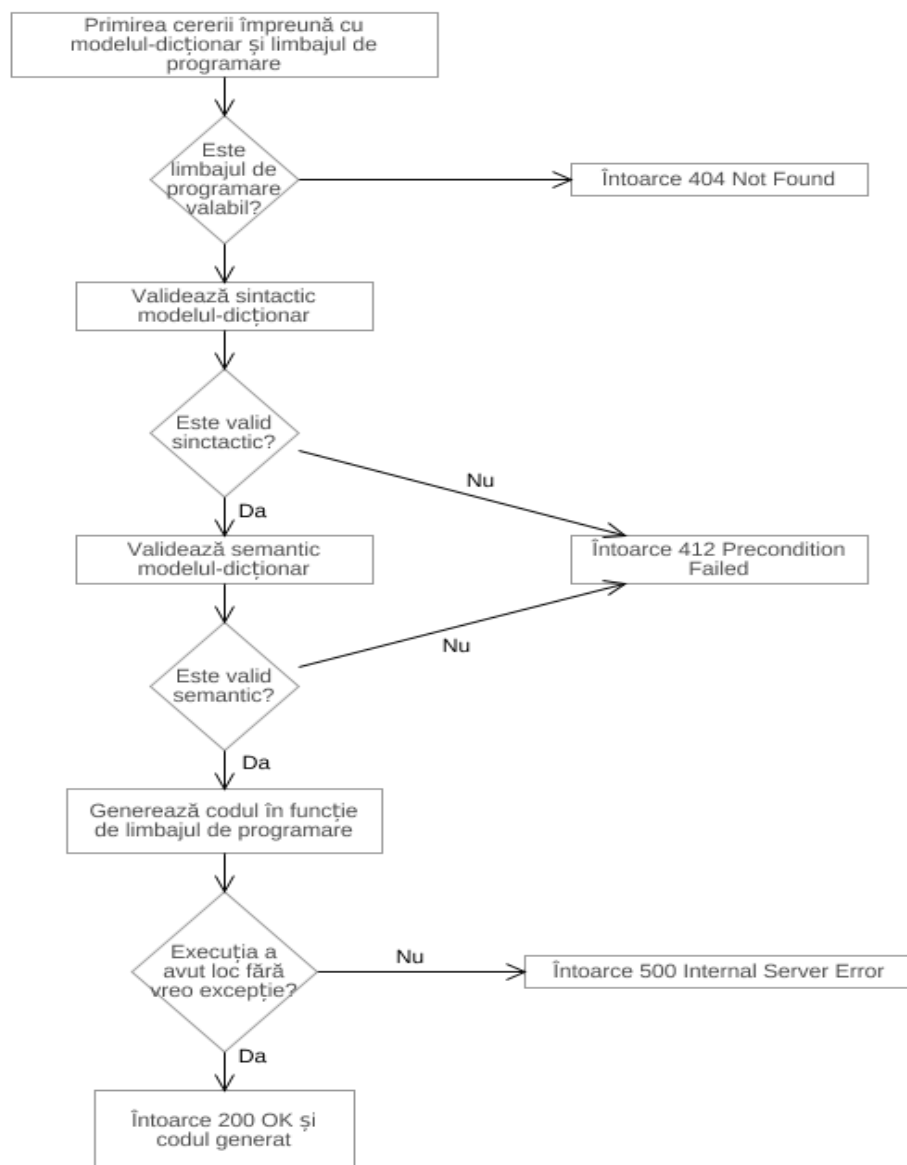


Figura 3.7. Diagrama fluxului serviciului de generare de cod

În continuare, se validează textul procesat primit la intrare în funcție de criteriile de validare, în două etape distincte:

- ❖ Etapa de validare sintactică, unde șirurile de caractere care sunt valori în modelul-dicționar trec printr-o serie de validări dictate de expresii regulate specifice. Acestea sunt enumerate și explicate în capitolul 4.2.
- ❖ Etapa de validare semantică, unde sunt semnalate erorile relative la logica modelului primit.

Validarea semantică ia în considerare următoarele greșeli de modelare:

- crearea mai multor clase cu același nume,
- crearea unei clase cu un nume invalid (de exemplu, tipuri de date predefinite, precum "int"),
- crearea unei clase ce se moștenește pe sine sau moștenește o clasă inexistentă în model,
- existența unei variabile cu tip necunoscut,
- existența a două sau mai multe variabile cu același nume în cadrul aceleiași clase,
- existența a două sau mai multe metode cu aceeași semnătură în cadrul aceleiași clase,
- existența unei metode ce întoarce un tip necunoscut,
- existența unei metode sau a unui constructor ce are doi sau mai mulți parametri cu același nume,
- existența unei metode sau a unui constructor ce are un parametru cu tip necunoscut.

Validitatea tipurilor se referă la existența acestora în model sau într-o listă de tipuri predefinite din multiple limbaje de programare, precum "string" sau "byte".

Fluxul continuă cu generarea codului. Deoarece metoda prin care se generează codul trebuie să fie una extensibilă relativ la limbajele de programare, am ales ca structura generatorului de cod să fie dependentă doar de structura modelului-dicționar.

În ideea extensibilității, considerăm că există câte un generator de cod pentru fiecare limbaj de programare adăugat în serviciu. Pentru a impune restricții în comportamentul fiecărui generator, există o interfață pe care orice generator trebuie să o implementeze.

Această interfață impune convertirea fiecărui element modelabil (clasă, variabilă, metodă sau parametru) într-un șir de caractere. Deci, pentru fiecare element există o anumită metodă ce trebuie implementată.

Structura interfeței generator este descrisă în următoarele rânduri.

```
public interface IGenerator
{
    Dictionary<string, string>
    GenerateProcessedText(ProcessedText processedText);
    string GenerateEntity(Entity entity);
    string GenerateVariable(Variable variable);
    string GenerateMethod(Method method, bool
    isEntityInterface);
    string GenerateParameter(Parameter parameter);
    string GenerateConstructor(Constructor
    constructor, string entityName);
}
```

Tipurile de date din fragmentul de cod prezentat sunt descrise mai sus, în Figura 3.5.

După ce fiecare entitate este convertită în codul asociat modelului său, acestea este plasată într-un dicționar. Acest dicționar are drept cheie numele fiecărei entități și drept valoare codul generat asociat respectivei entități.

Pentru a înțelege mai bine fluxul, vom recurge la un exemplu. Presupunem că se efectuează o cerere la serviciul de generare de cod în care avem în corp modelul-dicționar generat de procesorul de text în exemplul de la sfârșitul capitolului 3.1 și limbajul de programare "Java".

În primul pas se caută compatibilitatea cu limbajul de programare. Odată găsit, se inițializează obiectul JavaGenerator. După etapa de validare, exemplul este considerat analizabil deoarece modelul este construit corect de procesorul de text.

La faza de generare, codul rezultat arată astfel:

```
{
    "Bob": "
class Bob {
    Toyota car;
    public void drive() {
        throw new NotImplementedException();
    }
} ",
    "Toyota": "
class Toyota {
} "
}
```


3.3. Serviciul de integrare

Serviciul de integrare are rolul de a coordona interacțiunea dintre serviciul de procesare de text și serviciul de generare de cod. Acesta, primind un text și un limbaj de programare, comunică cu cele două servicii pentru a crea fluxul complet al aplicației, oferind codul ce modelează textul de la intrare. Acesta este descris prin Figura 3.8.

În cadrul acestui serviciu se pot înlocui modulele folosite cu ușurință, în vederea asigurării extensibilitatea aplicației. Mai multe detalii se regăsesc în capitolul 4.3.

Furnizăm un exemplu.

Către serviciul de integrare se trimite o cerere ce are în corp această structură:

```
{  
  
    "text": "Bob has a car. Bob's car is a  
Toyota. Bob can drive.",  
  
    "language": "java"  
}
```

Acesta trimite o cerere HTTP către serviciul de procesare de text cu valoarea cheii "**text**" din intrare. Primind înapoi textul procesat de la de la sfârșitul capitolului 3.1, se emite o cerere către serviciul de generare de cod împreună cu limbajul de programare specificat în cheia "**language**". În răspunsul de la serviciul de generare de cod se găsește codul sursă ce va constitui răspunsul serviciului de integrare.

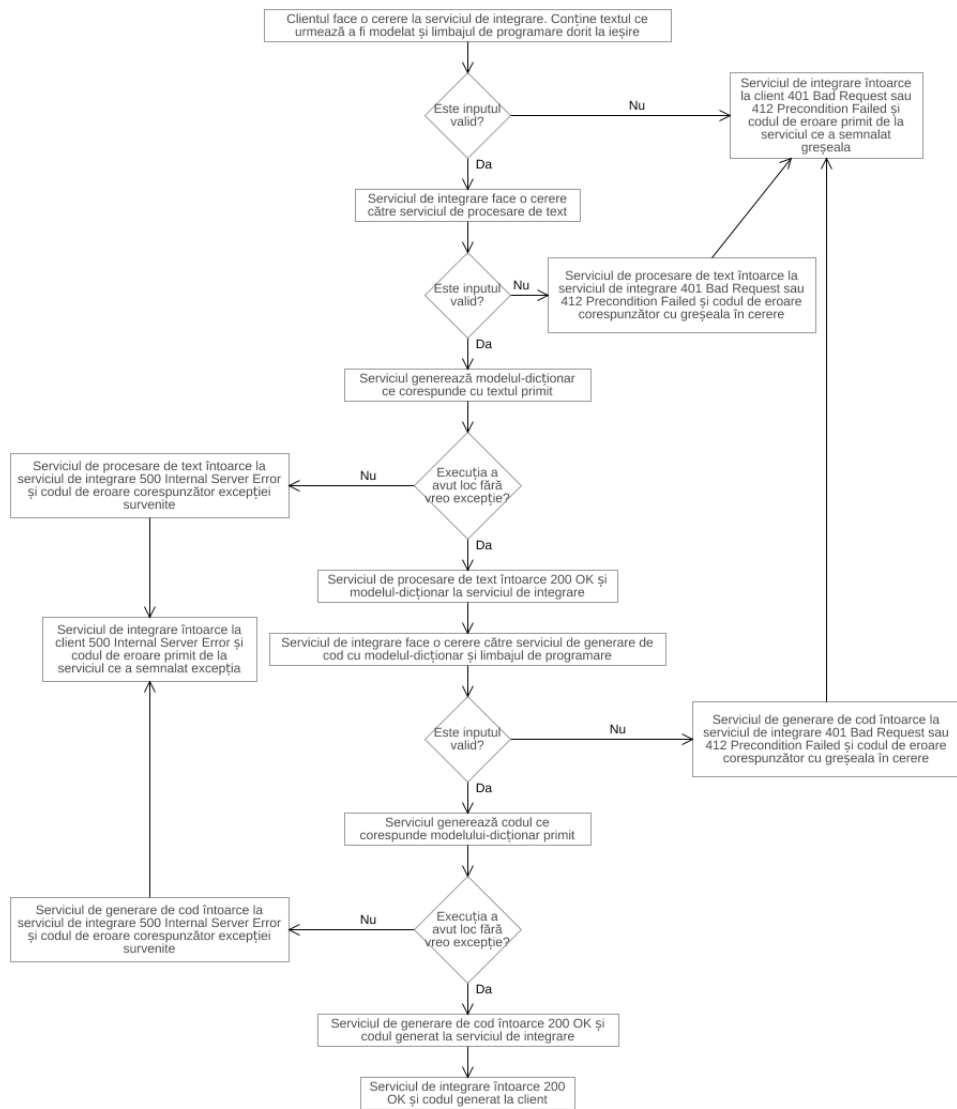


Figura 3.8. Diagrama fluxului serviciului de integrare

4. Detalii de implementare

În cadrul acestui capitol vom discuta despre tehnologiile folosite în cadrul fiecărui serviciu și anumite particularități de implementare. Precum am procedat și la arhitectura aplicației, vom trece prin fiecare serviciu în parte.

4.1. Detaliile serviciului de procesare de text

Serviciul de procesare a textului a recurs la Python [9] ca tehnologie primară. Această decizie a fost luată datorită ușurinței de a lucra cu structuri de tip dicționar, ușurinței de a lucra cu șiruri de caractere și datorită existenței bibliotecilor NLTK [2] și Bottle [13]. Uzul acestora va fi detaliat în acest capitol. De asemenea, vom descrie fluxul de preprocesare a textului (înainte de a fi analizat și prelucrat de analizatorul gramatical), vom explica prescurtările părților de vorbire utilizate în gramatică și vom descrie algoritmul de prelucrare a arborilor creați de analizatorul gramatical.

Mai întâi ne vom referi la modalitatea de a expune unealta de procesarea de text ca serviciu web. Am realizat acest lucru cu ajutorul bibliotecii Bottle [13] și a metodei POST din cadrul protocolului HTTP.

În următoarea secvență de cod este descrisă modalitatea de lucru cu biblioteca Bottle.

```
@route('/textProcessor', method='POST')
def text_preprocessor_handler():
    text=request.body.read().decode('UTF-8')
    processed_text = process(text)
    return processed_text
```

În continuare vom discuta despre pașii de preprocesare a textului. Aceștia sunt:

- ❖ Împărțirea textului în propoziții și împărțirea propozițiilor în cuvinte. Acești pași sunt necesari deoarece analizatorul gramatical poate procesa doar câte o listă de cuvinte o dată.
- ❖ Transformarea fiecărui cuvânt în forma sa din dicționar. Acest pas scade din complexitatea unei gramatici ce dorește să controleze un text. Spre exemplu, pentru a exprima moștenirea este suficientă combinația "be a". În cazul în care nu am fi adus cuvântul "be" la forma sa din dicționar, trebuiau incluse toate formele verbului "to be" în gramatică.
- ❖ Înlocuirea cuvintelor care nu sunt cuvinte-cheie cu partea acestora de vorbire. Această operație scade complexitatea gramaticii, deoarece nu mai este nevoie folosirea unei expresii regulate care să valideze cuvintele care nu sunt cheie, ele fiind verificate doar după partea lor de vorbire.

Biblioteca NLTK [2] este o unealtă de procesare a limbajului natural. Aceasta a oferit ajutor în partea de preprocesare a textului și crearea unui analizator gramatical pe baza gramaticii propuse.

În următorul tabel vom puncta fiecare componentă a bibliotecii folosită în cadrul serviciului și scopul acesteia în procesare.

Componenta utilizată	Scopul utilizării
sent_tokenizer	Împărțirea în propoziții a textului primit

word_tokenizer	Împărțirea în cuvinte a propozițiilor
pos_tag	Găsirea părților de vorbire a cuvintelor
WordNetLemmatizer	Găsirea formei din dicționar a cuvintelor
CFG	Validarea gramaticii folosite
BottomUpChartParser	Analizatorul sintactic pentru gramatica folosită

Tabel 4.1. Componente utilizate din cadrul bibliotecii NLTK

WordNetLemmatizer are nevoie de partea de vorbire a cuvântului pentru care caută forma din dicționar sub forma specificată în nltk.WordNet, dar pos_tag întoarce forma specificată în [14]. Din acest motiv am implementat o funcție care realizează conversia între cele două formate.

```
def pos_to_wordnet(treebank_pos):
    if treebank_pos.startswith('J'):
        return wordnet.ADJ
    if treebank_pos.startswith('V'):
        return wordnet.VERB
    if treebank_pos.startswith('R'):
        return wordnet.ADV
    return wordnet.NOUN
```

După preprocesare, fiecare propoziție este trecută prin analizatorul gramatical ce generează arbori propoziționali. Algoritmul de creare și prelucrare a arborilor este responsabil cu transformarea dintr-o structură greu-prelucrabilă și dependentă de biblioteca NLTK (nltk.Tree) într-o structură de tip dicționar din care pe urmă se extrage modelul-dicționar. Tot în cadrul acestui proces cuvintele care au fost înlocuite cu partea lor de vorbire sunt aduse la forma lor originală. Acest lucru este posibil deoarece concatenarea cuvintelor din frunzele arborelui crează propoziția de dinaintea procesării. În secvența următoare de cod regăsim algoritmul de procesare al arborilor de propoziție.

```
def tree_to_dict(tree, current_sentence_index,
                 current_position_in_sentence=0):
    tree_dict = dict()
    for t in tree:
        if isinstance(t, Tree) and isinstance(t[0], Tree):
            tree_dict[t.label()], current_position_in_sentence =
                tree_to_dict(t, current_sentence_index,
                             current_position_in_sentence)
        elif isinstance(t, Tree):
            if str(t[0]) in pos_list:
                tree_dict[t.label()] =
                    lemmatized[current_sentence_index]
                        [current_position_in_sentence]
            else:
                tree_dict[t.label()] = str(t[0])
                current_position_in_sentence += 1
    return tree_dict, current_position_in_sentence

def lemmatized_pos_terminals_to_processed_text
    (lemmatized_pos_terminals):
    processed_text = []
    for i in range(0, len(lemmatized_pos_terminals)):
        processed_sentence_tree =
            grammar_parser.parse(lemmatized_pos_terminals[i])
        processed, last_index = tree_to_dict(
            processed_sentence_tree, i)
        processed_text.append(processed['sentence'])
    return processed_text
```

4.2. Detaliile serviciului de generare de cod

În cadrul serviciului de generare de cod avem o abordare bazată pe platforma .NETCore [15]. Aceasta facilitează validarea sintactică a schemei-model cu ajutorul DataAnnotations, un modul extensibil, simplu de folosit.. De asemenea .NETCore ajută și la expunerea de servicii web prin extinderea clasei Controller din spațiul de nume.AspNetCore.Mvc.

DataAnnotations permite dezvoltatorului să folosească validări pentru valorile variabilelor unei clase. De asemenea se pot implementa adnotări de validare proprii.

Vom enumera validările de tip DataAnnotations pentru fiecare element al modelului-dicționar în tabelul următor.

Element validat	Validare de tip DataAnnotations
Modificator de acces	<pre>[RegularExpression(" ^public\$ ^protected\$ ^private\$ ^\$ ")]</pre>

Nume de entitate Nume de variabilă Nume de metoda Nume de parametru	[NotKeyword] [Required] [RegularExpression(" [_a-zA-Z][_a-zA-Z0-9]* ")]
Tipul de entitate	[RegularExpression(" ^class\$ ^interface\$")]
Nume de entitate moștenită	[NotKeyword] [RegularExpression(" [_a-zA-Z][_a-zA-Z0-9]* ")]
Tip de variabilă Tip de parametru Tip de întoarcere al unei metode	[RegularExpression(" [_a-zA-Z][_a-zA-Z0-9]* ")]

Tabel 4.2. Validările șirurilor de caractere din modelul-dicționar

Atributul `[NotKeyword]` este un atribut implementat de noi pentru a impune anumitor șiruri de caractere să nu poată lua valori ce sunt considerate cuvinte cheie în limbajul de programare în care urmează să generăm cod (de exemplul, în Java nu putem defini o clasă cu numele "**final**").

Interfața **IGenerator** definită la capitolul 3.2 are două implementări, una pentru limbajul **Java** și una pentru limbajul **C#**. Există multiple diferențe la implementare între acestea două; spre exemplu, pentru a declara o variabilă constantă, în **Java** folosim cuvântul cheie **final**, pe când în **C#** folosim cuvântul cheie **const**. La fel și pentru proprietatea unei clase de a fi finale (în **Java** este **final**, în **C#** este **sealed**), sau tipul de șiruri de caractere (în **Java** este **String**, în **C#** este **string**).

4.3. Detaliile serviciului de integrare

Serviciul de integrare are rolul de a concatena fluxurile celorlalte două servicii. Acesta există în ideea extensibilității (în cazul în care un serviciu trebuie înlocuit, acesta doar trebuie implementat și atașat în cadrul serviciului de integrare).

Sistemul de gestionare a serviciilor folosite este facilitat de fișierul `server_utilities.json`. Acesta are următoarea structură.

```
{
  "languages": {
    "java":
      "http://localhost:4000/codeGenerator/java",
    "c#":
      "http://localhost:4000/codeGenerator/csharp",
  },
  "text_processor_uri":
    "http://localhost:3000/textProcessor",
  "required_input_fields": [
    "text",
    "language"
  ]
}
```

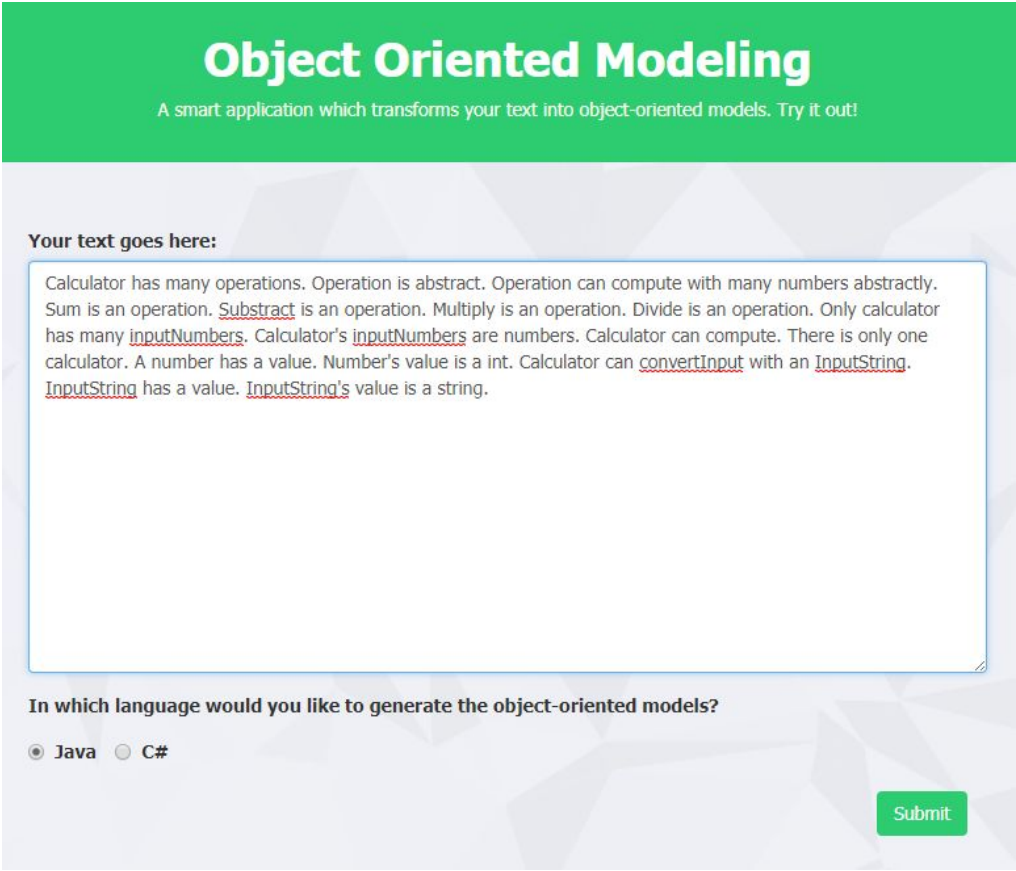
Cu ajutorul acestui fișier serviciul de integrare cunoaște ce limbaje sunt disponibile pentru generarea de cod, cunoaște toate locațiile unde trebuie să trimită cereri HTTP pentru a comunica cu celelalte servicii și știe de ce informații are nevoie la intrare pentru a completa fluxul.

5. Exemple de utilizare

În cadrul acestui capitol vom exemplifica utilizarea serviciilor descrise într-un caz practic. Vom exemplifica perechile intrare-ieșire cu ajutorul unei interfețe grafice.

Exemplul 1: Dorim să modelăm un calculator. Acesta trebuie să poată primi o mulțime de numere și să poată efectua operațiile de bază cu ele. Acesta trebuie să poată primi la intrare un șir de caractere. Calculatorul trebuie să respecte șablonul de proiectare Singleton și să fie implementat în Java.

Textul ce modelează problema:



The screenshot shows a web application titled "Object Oriented Modeling". Below the title is a subtitle: "A smart application which transforms your text into object-oriented models. Try it out!". There is a text input area with the placeholder "Your text goes here:". Inside this area, there is a sample text: "Calculator has many operations. Operation is abstract. Operation can compute with many numbers abstractly. Sum is an operation. Subtract is an operation. Multiply is an operation. Divide is an operation. Only calculator has many inputNumbers. Calculator's inputNumbers are numbers. Calculator can compute. There is only one calculator. A number has a value. Number's value is a int. Calculator can convertInput with an InputString. InputString has a value. InputString's value is a string." Below the text area is a question: "In which language would you like to generate the object-oriented models?". There are two radio buttons: "Java" (selected) and "C#". At the bottom right is a green "Submit" button.

Figura 5.1. Modelarea problemei folosind text natural controlat

În continuare regăsim codul rezultat:



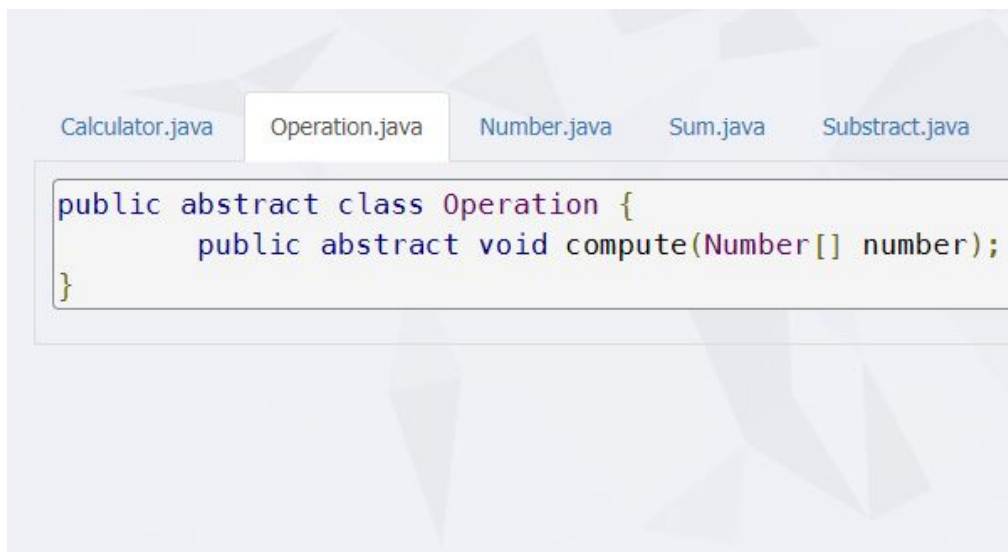
Object Oriented Mo
A smart application which transforms your text into object-or

Your text: Calculator has many operations. Operation is abstract. Operation can comput operation. Multiply is an operation. Divide is an operation. Only calculator has Calculator can compute. There is only one calculator. A number has a value. InputString. InputString has a value. InputString's value is a string.

Calculator.java Operation.java Number.java Sum.java Subtract.java Multiply.java

```
public class Calculator {
    public Operation[] operation;
    private Number[] inputnumbers;
    private static Calculator instance;
    private Calculator() {
    }
    public void compute() {
        throw new UnsupportedOperationException();
    }
    public static Calculator getInstance() {
        throw new UnsupportedOperationException();
    }
    public void convertinput(InputString inputstring) {
        throw new UnsupportedOperationException();
    }
}
```

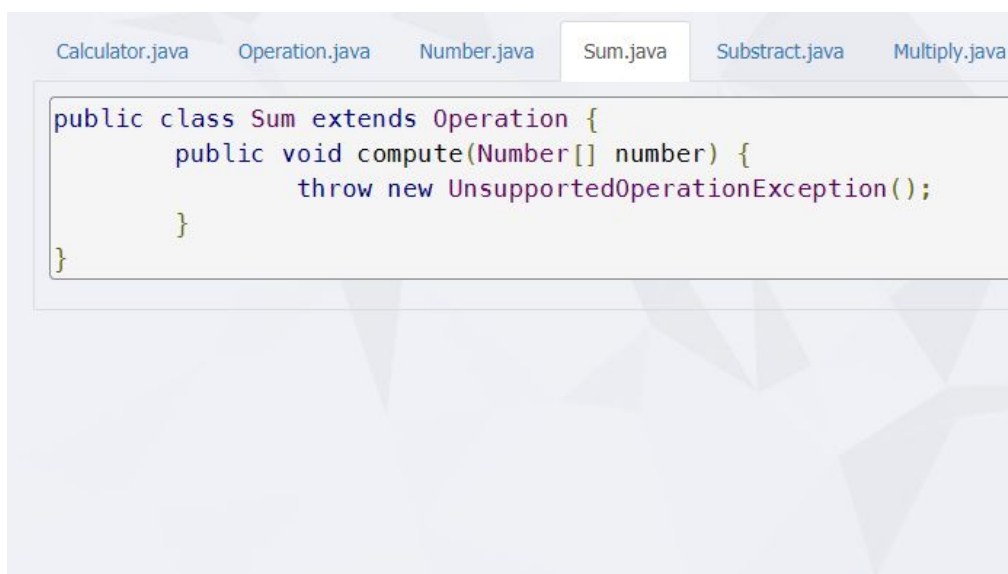
Figura 5.2. Codul rezultat pentru clasa *Calculator*



The screenshot shows an IDE with a tab bar at the top containing five tabs: Calculator.java, Operation.java, Number.java, Sum.java, and Subtract.java. The Operation.java tab is selected. The code editor displays the following Java code:

```
public abstract class Operation {  
    public abstract void compute(Number[] number);  
}
```

Figura 5.3. Codul rezultat pentru clasa *Operation*



The screenshot shows an IDE with a tab bar at the top containing six tabs: Calculator.java, Operation.java, Number.java, Sum.java, Subtract.java, and Multiply.java. The Sum.java tab is selected. The code editor displays the following Java code:

```
public class Sum extends Operation {  
    public void compute(Number[] number) {  
        throw new UnsupportedOperationException();  
    }  
}
```

Figura 5.4. Codul rezultat pentru clasa *Sum*

Exemplul 2: Dorim să modelăm Modelul-Dicționar descris la capitolul 3.1.

Textul ce modelează problema:

An entity has many variables. An entity has many methods. An entity has many constructors. An entity has a name. Entity's name is a string. An entity has a type. Entity's type is a String. Entity has a accessModifier. Entity's accessModifier is a String. Entity has an isSealed. Entity's IsSealed is a bool. Entity has an isAbstract. Entity's isAbstract is a bool. Entity has an inherits. Entity's inherits is a String. Variable has a name. Variable has a Type. Variable has an IsCollection. Variable has an accessModifier. Variable has an isConst. Variable has an isStatic. Method has a name. A method has a ReturnedType. A method has an AccessModifier. A method has an IsSealed. A method has an IsStatic. A method has an IsAbstract. A method has many parameters. A parameter has a name. A parameter has a type. A parameter has a IsCollection. A constructor has an accessModifier. Variable's name is a String. Variable's type is a string. Variable's IsCollection is a bool. Variable's accessModifier is a string. Variable's isConst is a bool. Variable's isStatic is a bool. Method's name is a string. Method's ReturnedType is a string. Method's accessModifier is a string. Method's IsSealed is a bool. Method's IsStatic is a bool. Method's isAbstract is a bool. Parameter's name is a string. Parameter's type is a string. Parameter's isCollection is a bool. Constructor's accessModifier is a string. Constructor has many parameters.

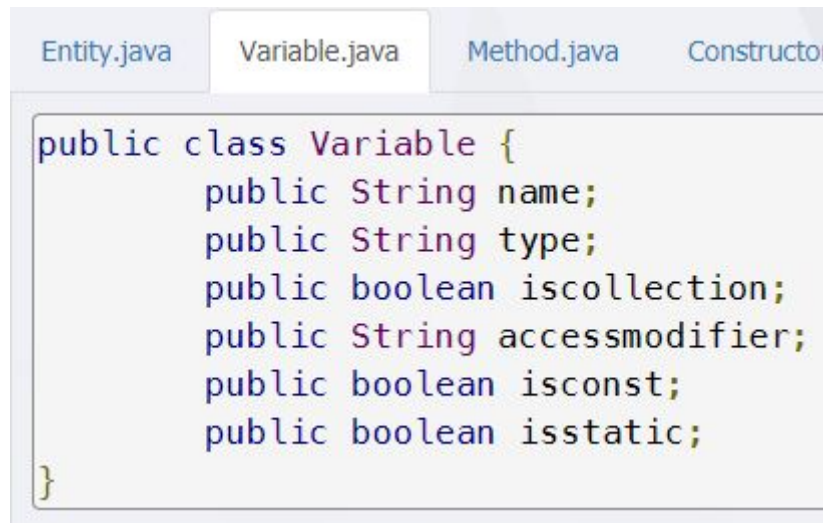
Codul rezultat:



```
Entity.java  Variable.java  Method.java  Constructor.java

public class Entity {
    public Variable[] variable;
    public Method[] method;
    public Constructor[] constructor;
    public String name;
    public String type;
    public String accessmodifier;
    public boolean issealed;
    public boolean isabstract;
    public String inherits;
}
```

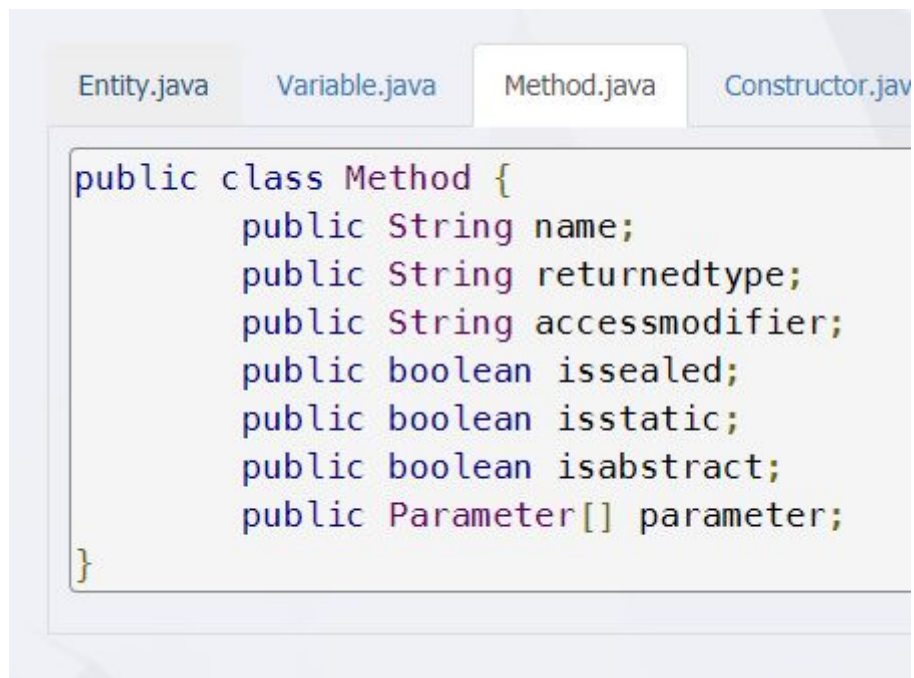
Figura 5.5. Codul rezultat pentru clasa *Entity*



The screenshot shows a code editor with four tabs: Entity.java, Variable.java (selected), Method.java, and Constructor.java. The Variable.java tab contains the following Java code:

```
public class Variable {  
    public String name;  
    public String type;  
    public boolean iscollection;  
    public String accessmodifier;  
    public boolean isconst;  
    public boolean isstatic;  
}
```

Figura 5.6. Codul rezultat pentru clasa *Variable*



The screenshot shows a code editor with four tabs: Entity.java, Variable.java, Method.java (selected), and Constructor.java. The Method.java tab contains the following Java code:

```
public class Method {  
    public String name;  
    public String returnedtype;  
    public String accessmodifier;  
    public boolean issealed;  
    public boolean isstatic;  
    public boolean isabstract;  
    public Parameter[] parameter;  
}
```

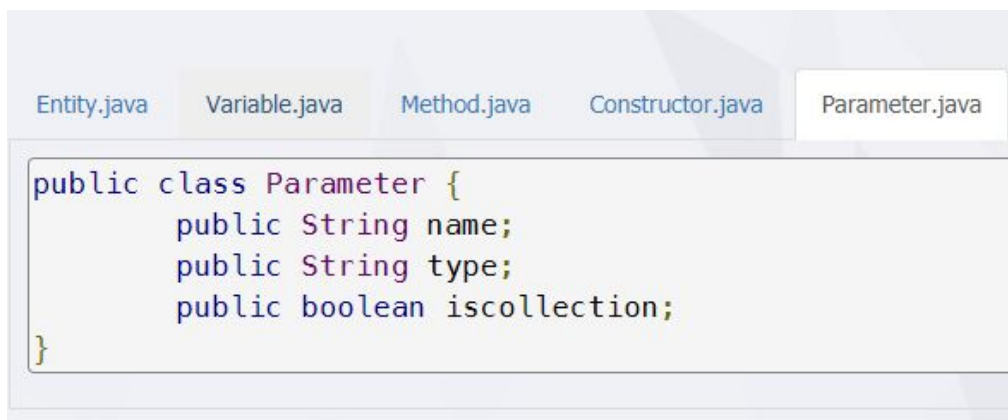
Figura 5.7. Codul rezultat pentru clasa *Method*



```
Entity.java Variable.java Method.java Constructor.java

public class Constructor {
    public String accessmodifier;
    public Parameter[] parameter;
}
```

Figura 5.8. Codul rezultat pentru clasa *Constructor*



```
Entity.java Variable.java Method.java Constructor.java Parameter.java

public class Parameter {
    public String name;
    public String type;
    public boolean iscollection;
}
```

Figura 5.8. Codul rezultat pentru clasa *Parameter*

6. Direcții de viitor

Ca extensii ale aplicației ne putem gândi la situații precum:

- ❖ Generare de diagrame de clase din modelul-dicționar. Acest lucru este posibil datorită implementării acestei aplicații într-o manieră extensibilă.
- ❖ Crearea unei abordări bazată pe micro-servicii, în care fiecare pas de procesare a textului și crearea și validarea modelului-dicționar este un serviciu independent.
- ❖ Crearea fluxului invers: primind o secvență de cod, să putem extrage din aceasta modelul, pe care să îl expunem ca limbaj natural.
- ❖ Aplicarea în alte contexte diverse, spre exemplu, extragerea textului natural dintr-o conversație (fie pe cale orală, fie pe cale scrisă) și expunerea acestuia ca model orientat obiect.
- ❖ Adăugarea mai multor limbaje de programare prin crearea de mai multe implementări ale interfeței **IGenerator**.
- ❖ Adăugarea de diverse metode de a controla textul prin crearea mai multor gramatici; spre exemplu, o gramatică ce controlează un text în limba română.

Concluziile lucrării

În cadrul acestei lucrări am discutat despre conceperea unei aplicații ce convertește un limbaj natural controlat într-un limbaj de programare.

Pentru a realiza acest lucru am propus o arhitectură bazată pe trei servicii web: un serviciu de procesare de text, un serviciu de generare de cod și un serviciu de integrare.

Primul serviciu, printr-o serie de operații de procesare de text, reușește să convertească un text scris într-un limbaj natural controlat într-o formă independentă de limbajul de programare. Această structură de date se numește *modelul-dicționar* și este propusă în această lucrare ca schemă de modelare pentru conceptele orientării obiect.

Serviciul de generare de cod primește date în forma propusă prin modelul-dicționar și le validează relativ la corectitudinea în cadrul limbajelor de programare în care, pe urmă, generează cod.

Aplicația a fost construită într-o manieră ce vizează extensibilitatea, existând un serviciu de integrare ce îmbină fluxurile celor două servicii descrise mai sus, folosind o gramatică pentru a valida textul natural, folosind o structură de date independentă de limbaj pentru a modela conceptele orientării obiect și prin existența unei interfețe de generare de cod pentru extensibilitatea relativă la limbajul de programare produs.

Această aplicație poate servi ca instrument util programatorilor pentru a genera clasele pe care vor să le implementeze pe urmă, poate ajuta persoanele nu foarte experimentate în informatică pentru a înțelege anumite concepte și poate facilita conversațiile între persoanele familiarizate cu conceptele orientării obiect și cele ce nu le știu.

Bibliografie

- [1] G. Booch, *Object-oriented analysis and design with applications (2nd ed.)*, Addison Wesley Longman, Inc, 1994
- [2] NLTK, <http://www.nltk.org/>
- [3] AlchemyAPI, IBM Watson Developer Cloud, <https://www.ibm.com/watson/alchemy-api.html>
- [4] YUML, <https://yuml.me/diagram/scruffy/class/samples>
- [5] Fluent Editor, <http://www.cognitum.eu/semantics/FluentEditor/>
- [6] S. Buraga, *Web Applications Development*, prezentarea *Specifying Ontologies via OWL 2. Description. Logics. Reasoning*, slide #163, 2017, Facultatea de Informatică Iași
- [7] Blockly, Google, <https://developers.google.com/blockly/>
- [8] JSON Schema, <http://json-schema.org/>
- [9] Python programming language, <https://www.python.org/>
- [10] Servicii Web, <https://www.w3.org/TR/ws-gloss/>
- [11] Singleton, <https://www.gofpatterns.com/creational-design-patterns/creational-patterns/singleton-pattern.php>
- [12] S. Bird, E. Klein, E. Loper, *Natural Language Processing with Python*, chapter 8.3, <http://www.nltk.org/book/ch08.html>,
- [13] Bottle, <https://bottlepy.org/docs/dev/>
- [14] Penn Treebank Tag Set, http://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html
- [15] .NET Core, <https://www.microsoft.com/net/core>