DictionaryManagement

Student: Maxim Bogdan-Gheorghe

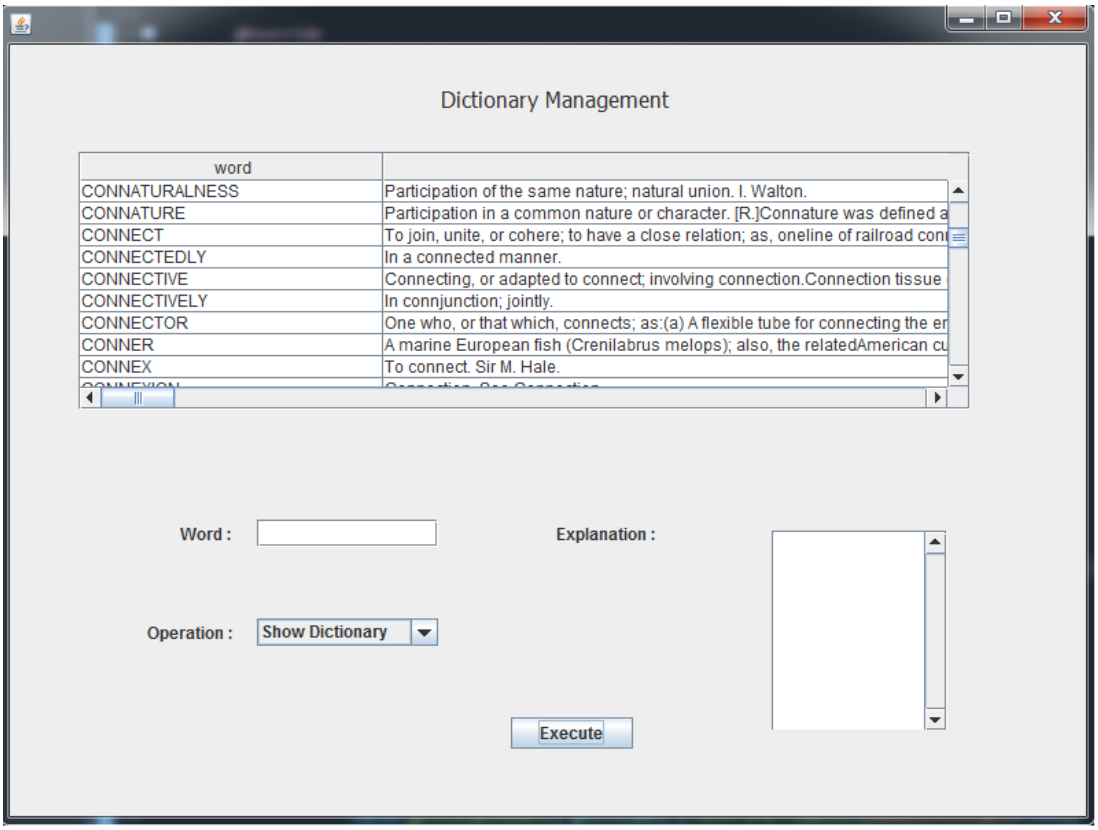
UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Cuprins

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

1. Obiectivul temei

Obiectivul temei este proiectarea, implementarea și realizarea unei aplicații menite pentru a gestiona un dictionar al limbii engleze. Aplicația poate fi folosită atât de un specialist, cât și de o persoana obijnuită care dorește să se documenteze asupra unor definiții generale al unor cuvinte din limba engleză. De asemenea, utilizatorul poate adăuga cuvinte noi, șterge cuvinte vechi, precum și salva și încărca starea dicționarului.



UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

2.Analiza problemei,modelare,scenarii,cazuri de utilizare

Primul pas efectuat în analiza și modelarea problemei este cel de a identifica substantivele principale din cerință pentru a putea stabili clasele utilizate. Dicționarul este cel mai general substantiv care poate fi reprezentat drept clasă, prin urmare acesta va fi transpus primul. Cuvintele vor fi păstrate într-o asociere cu definițiile acestora, iar structura de date utilizată pentru a memora aceste asocieri va fi principala structură de date modificabilă. Din clasa Dictionary se va facilita accesul direct asupra acesteia. Pentru a permite extinderea rapidă a operațiilor se va folosi șablonul de proiectare : "decorator". Astfel, creăm clasa abstractă OperationDecorator care are ca rol menținerea unei referințe către dicționarul propriu-zis și efectuarea operațiilor asupra acestuia.
În continuare, proiectăm clase pentru restul operațiilor :

-AddWord

-RemoveWord

-WordMatcher

-PopulateDictionary

-SaveDictionary

-LoadDictionary

Clasa AddWord are rolul de a adăuga un cuvânt în dicționar. Clasa RemoveWord are rolul de a șterge un cuvânt din dicționar. Clasa WordMatcher va găsi un cuvânt în dicționarul dat după un anumit șablon. PopulateDictionary va analiza un fișier JSON din care va extrage un șir de caractere ce reprezintă un dicționar deja format din cuvinte și definiții. Aceasta va încărca apoi clasa Dictionary cu șirul nou format. Clasa SaveDictionary va salva dicționarul curent în fișierul dicstate.ser, iar clasa LoadDictionary va încărca din fișierul respectiv. Astfel, operațiile de bază efectuate pe dicționar sunt îmbuătățite prin adăugarea de operații noi. De asemenea, creerea și implementarea altor operații necesită un timp suplimentar minim, precum și o putere de muncă limitată. Ca și tehnici, am folosit design by contract pentru a realiza un anumit contract cu utilizatorul unei metode, serializare pentru a salva și încărca starea unui obiect și analizare și retragere de informații dintr-un fișier JSON pentru a popula dicționarul. Scenariile de utilizare pot varia, atât pentru oameni de rând, cât și pentru specialiști. De asemenea, programul poate fi transformat foarte ușor în orice tip de dicționar prin ștergerea dicționarului curent și introducerea

termenilor specifici. De asemenea softul se poate utiliza și în școli, pentru a ajuta elevii/studenții atât în căutarea lor de cuvinte noi, cât și în personalizarea unui vocabular propriu care poate fi implementat cu ușurință datorită flexibilității șablonului de proiectare ales.

3.Proiectare

Pentru prezentarea proiectării aplicației, continuăm prin a prezenta tehnicile de programare utilizate :

Design by contract :

- reprezinta un "contract" care specifică restricțiile la care trebuie să se supună datele de intrare ale unei metode, valorile posibile de ieșire si stările în care se poate afla programul - aceste restrictii sunt date sub forma unor:

a) precondiții: reprezintă obligațiile pe care datele de intrare ale unei metode trebuie sa le respecte pentru ca metoda sa functioneze corect

b) postconditii: reprezinta garantiile pe care datele de iesire ale unei metode le ofera

c) invarianti: reprezinta conditii impuse starilor in care programul se poate afla la un moment dat

Cel care a implementat clasa su metoda restectiva ii spune utilizatorului ce date sunt considerate valide ca date de intrare. Aceasta il scuteste sa foloseasca teste de validare a datelor, care in unele cazuri doar ar incetini algoritmul, ele fiind redundante cu teste care deja sunt facute pentru validarea datelor, de exemplu imediat ce au fost introduse de catre utilizator.

În acelasi timp utilizatorul stie din contract la ce date posibile de iesire sau stari intermediare sa se astepte si atunci poate sa-si optimizeze propriul cod in functie de acestea

Contractul unei clase sau al unei metode se specifica literar, printr-un text, inclus in program ca un comentariu sau scris in documentatia aferenta - in faza de dezvoltare a unui program se pot folosi instructiuni care sa testeze indeplinirea contractului, instructiuni care sa fie scoase pe urma (manual sau automat) din codul final.

O asertiune genereaza o exceptie speciala cu textul dat dupa ":", in cazul in care conditia ei este falsa. Deoarece asertiunile au fost introduse mai tarziu in limbajul Java trebuie specificat la optiunile de compilare: -source 1.4 - implicit asertiunile nu fac nimic (sunt dezactivate), ca si cum codul este pregatit pentru livrare. Daca dorim sa activam asertiunile trebuie sa specificam la optiuni: -enableassertions (sau -ea) .

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Serializare

Serializarea este o metoda prin care se pot salva, într-o maniera unitara, datele împreuna cu signatura unui obiect. Folosind aceasta operatie se poate salva într-un fisier, ca sir de octeti, o instanta a unei clase, în orice moment al executiei. De asemenea, obiectul poate fi restaurat din fisierul în care a fost salvat în urma unei operatii de serializare.

Utilitatea serializarii constă în următoarele aspecte:
Asigură un mecanism *simplu de utilizat* pentru salvarea și restaurarea a datelor.
Permite *persistența obiectelor*, ceea ce înseamna că durata de viața a unui obiect nu este determinată de execuția unui program în care acesta este definit - obiectul poate exista și între apelurile programelor care îl folosesc. Acest lucru se realizează prin serializarea obiectului și scrierea lui pe disc înainte de terminarea unui program, apoi, la relansarea programului, obiectul va fi citit de pe disc ¸si starea lui refacută. Acest tip de persistență a obiectelor se numește *persistența ușoară*, întrucât ea trebuie efectuată explicit de către programator ¸si nu este realizată automat de către sistem.

Compensarea diferențelor între sisteme de operare - transmiterea unor informații între platforme de lucru diferite se realizează unitar, independent de formatul de reprezentare a datelor, ordinea octeților sau alte detalii specifice sistemelor repective.

Transmiterea datelor în rețea - Aplicațiile ce rulează în rețea pot comunica între ele folosind fluxuri pe care sunt trimise, respectiv recepționate,obiecte serializate

RMI (Remote Method Invocation) - este o modalitate prin care metodele unor obiecte de pe o altă mașină pot fi apelate ca și cum acestea ar exista local pe mașina pe care rulează aplicația. Atunci când este trimis un mesaj către un obiect ”remote”

(de pe altă mașină), serializarea este utilizată pentru transportul argumentelor prin rețea și pentru returnarea valorilor.

Java Beans - sunt componente reutilizabile, de sine stătătoare ce pot fi utilizate în medii vizuale de dezvoltare a aplicațiilor. Orice componentă Bean are o stare definită de valorile implicite ale proprietăților sale, stare care este specificată în etapa de design a aplicației. Mediile vizuale folosesc mecanismul serializării pentru asigurarea persistenței componentelor Bean.

Un aspect important al serializării este că nu salvează doar imaginea unui obiect ci și toate referințele la alte obiecte pe care acesta le conține. Acesta este un proces recusiv de salvare a datelor, întrucât celelalte obiectele referite de obiectul care se serializează pot referi la rândul lor alte obiecte, și așa mai departe. Așadar referințele care construiesc starea unui obiect formează o întreagă rețea, ceea ce înseamnă că un algoritm general de salvare a stării unui obiect nu este

tocmai facil. In cazul în care starea unui obiect este formată doar din valori ale unor variabile de tip primitiv, atunci salvarea informațiilor încapsulate în acel obiect se poate face și prin salvarea pe rând a datelor, folosind clasa DataOutputStream, pentru ca apoi să fie restaurate prin metode ale clasei DataInputStream, dar, așa cum am vazut, o asemenea abordare nu este în general suficientă, deoarece pot apărea probleme cum ar fi: variabilele membre ale obiectului pot fi instanțe ale altor obiecte, unele câmpuri pot face referință la același obiect, etc.

Salvarea datelor înapsulate într-un obiect se poate face si prin salvarea pe rând a datelor, folosind clasa DataOutputStream, pentru ca apoi sa fie restaurate prin metode ale clasei DataInputStream, dar o asemenea abordare nu este în general suficienta, deoarece pot aparea probleme cum ar fi :

- datele obiectului pot fi instante ale altor obiecte
- în unele cazuri, este necesara si salvarea tipului datei
- unele câmpuri fac referinta la acelasi obiect

Asadar, prin serializare sunt surprinse atât datele, signatura clasei (numele metodelor si definitia lor - nu si implementarea) precum si starea obiectului.

Pentru a putea fi serializat un obiect trebuie sa fie instanta a unei clase care implementeaza una din interfetele :

- java.io.Serializable sau

- java.io.Externalizable (care extinde clasa Serializable)

Interfata Serialize nu are nici o metoda, ea da doar posibilitatea de a specifica faptul ca se doreste ca o anumita clasa sa poata fi serializata. Declaratia unei astfel de clase ar fi :

class ClasaSerializabila implements Serializable {...}

In urma serializarii obiectele sunt pot fi salvatr într-un fisier, în acelasi fisier putând fi salvate si mai multe obiecte. Operatiile de intrare iesire la nivelul obicetelor se realizeaza prin intermediul unor fluxuri de obiecte, implementate de clasele ObjectInputStream si ObjectOutputStream. Salvarea unui obiect într-un fisier se realizeaza astfel :

```
MyObject o = new MyObject();
FileOutputStream fout = new FileOutputStream("fisier");
ObjectOutputStream sout = new ObjectOutputStream(fout);
sout.writeObject(o);
```

Restaurarea unui obiect salvat într-un fisier se face într-o maniera asemanatoare:

```
FileInputStream fin = new FileInputStream(“fisier”);
ObjectInputStream sin = new ObjectInputStream(fin);
o = (MyObject) sin.readObject();
```

Pe lânga metodele de scriere/citire a obiectelor cele doua clase pun la dispozitie si metode pentru scrierea tipurilor de date primare, astfel încât apeluri ca cele de mai jos sunt permise :

```
FileOutputStream ostream = new FileOutputStream("t.tmp");
      ObjectOutputStream p = new ObjectOutputStream(ostream);
      p.writeInt(12345);
      p.writeObject("Today");
      p.writeObject(new Date());
      p.flush();
      ostream.close();

FileInputStream istream = new FileInputStream("t.tmp");
      ObjectInputStream p = new ObjectInputStream(istream);
      int i = p.readInt();
      String today = (String)p.readObject();
      Date date = (Date)p.readObject();
      istream.close();
```

`ObjectInputStream` si `ObjectOutputStream` implementeaza indirect interfetele `DataInput`, respectiv `DataOutput`, interfețe ce declara metode atât pentru scrierea/citirea datelor primitive, cât si pentru scrierea/citirea obiectelor. Pentru transferul obiectelor sunt folosite metodele:

```
final void writeObject( java.lang.Object obj )
     throws java.io.IOException
final java.lang.Object readObject( )
     throws java.io.OptionalDataException,
     java.lang.ClassNotFoundException, java.io.IOException
```

Acestea apeleaza la rândul lor metodele implicte de transfer `defaultWriteObject` si `defaultReadObject` (având aceleasi signaturi ca mai sus).

Personalizarea serializarii se realizeaza prin supradefinirea (într-o clasa serializabila!) a metodelor writeObject si readObject, modificând astfel actiunea lor implicita.
Metoda writeObject controleaza ce date sunt salvate si este uzual folosita pentru a adauga informatii suplimentare la cele scrise implicit de metoda defaultWriteObject.
Metoda readObject controleaza modul în care sunt restaurate obiectele, citind informatiile salvate si, eventual, modifcând starea obiectelor citite astfel încât ele sa corespunda anumitor cerinte.
Aceste metode trebuie obligatoriu sa aiba urmatorul format:
private void writeObject(ObjectOutputStream stream)
throws IOException
private void readObject(ObjectInputStream stream)
throws IOException, ClassNotFoundException
De asemenea, uzual, primul lucru pe care trebuie sa îl faca aceste metode este apelul la metodele standard de serializare a obiectelor defaultWriteObject, respectiv defaultReadObject si abia apoi sa execute diverse operatiuni suplimentare. Forma lor generala este:

```
private void writeObject(ObjectOutputStream s)
throws IOException {
s.defaultWriteObject();
// personalizarea serializarii
}
private void readObject(ObjectInputStream s)
throws IOException,ClassNotFoundException {
s.defaultReadObject();
// personalizarea deserializarii
. . .
// actualizarea starii obiectului (daca e necesar)
}
```

Metodele writeObject si readObject sunt responsabile cu serializarea clasei în care sunt definite, serializarea superclasei sale fiind facuta automat (si implicit). Daca însa o clasa trebuie sa-si coordoneze serializarea proprie cu serializarea superclasei sale, atunci trebuie sa implementeze interfata Externalizable.

Exista cazuri când dorim ca unele variabile membre sau sub-obiecte ale unui obiect sa nu fie salvate automat în procesul de serializare. Acestea sunt cazuri comune atunci când respectivele câmpuri reprezinta informatii confidentiale, cum ar fi parole, sau variabile auxiliare pe care nu are rost sa le salvam. Chiar declarate ca private în cadrul clasei aceste câmpuri participa la serializare. O modalitate de a controla serializare este implementarea interfetei Externalizable, asa cum am vazut anterior. Aceasta metoda este însa incomoda atunci când clasele sunt greu de serializat iar multimea câmpurilor care nu trebuie salvate este redusa.
Pentru ca un câmp sa nu fie salvat în procesul de serializare atunci el trebuie declarat cu modificatorul **transient** si trebuie sa fie ne-static. De exemplu, declararea unei parole ar trebui facuta astfel:
transient private String parola; //ignorat la serializare
<
Atentie
Modificatorul static anuleaza efectul modificatorului transient;

static transient private String parola; //participa la serializare
De asemenea, nu participa la serializare sub-obiectele neserializabile ale unui obiect, adica cele ale caror clase nu au fost declarate ca implementând interfata Serializable (sau Externalizable).
Exemplu: (câmpurile marcate 'DA' participa la serializare, cele marcate 'NU',
nu participa)
class A { ... }
class B implements Serializable { ... }
public class Test implements Serializable {
private int x; // DA
transient public int y; // NU
static int var1; // DA
transient static var2; // **DA**
A a; // NU
B b1; // DA
transient B b2; // NU
}
Atunci când o clasa serializabila deriva dintr-o alta clasa, salvarea câmpurilor clasei parinte se va face doar daca si aceasta este serializabila. In caz contrar, subclasa trebuie sa salveze explicit si câmpurile mostenite.

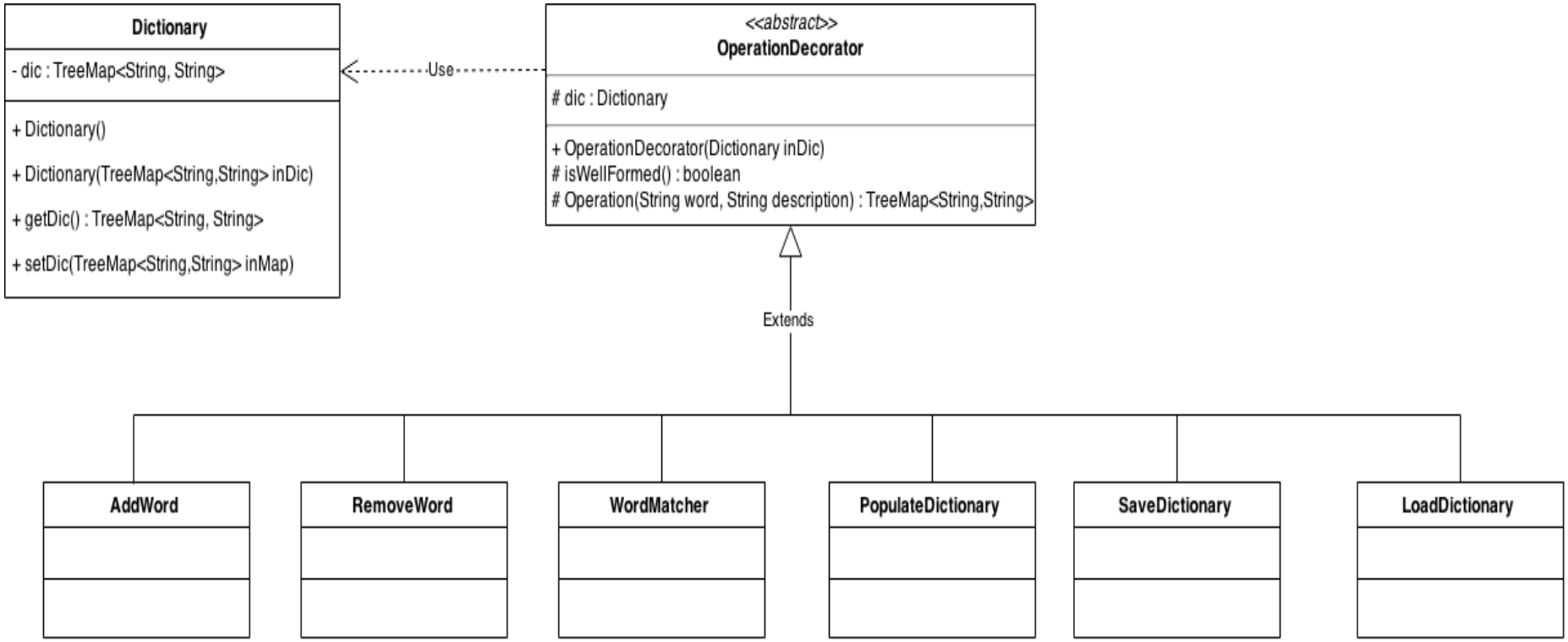
Șabloane de proiectare

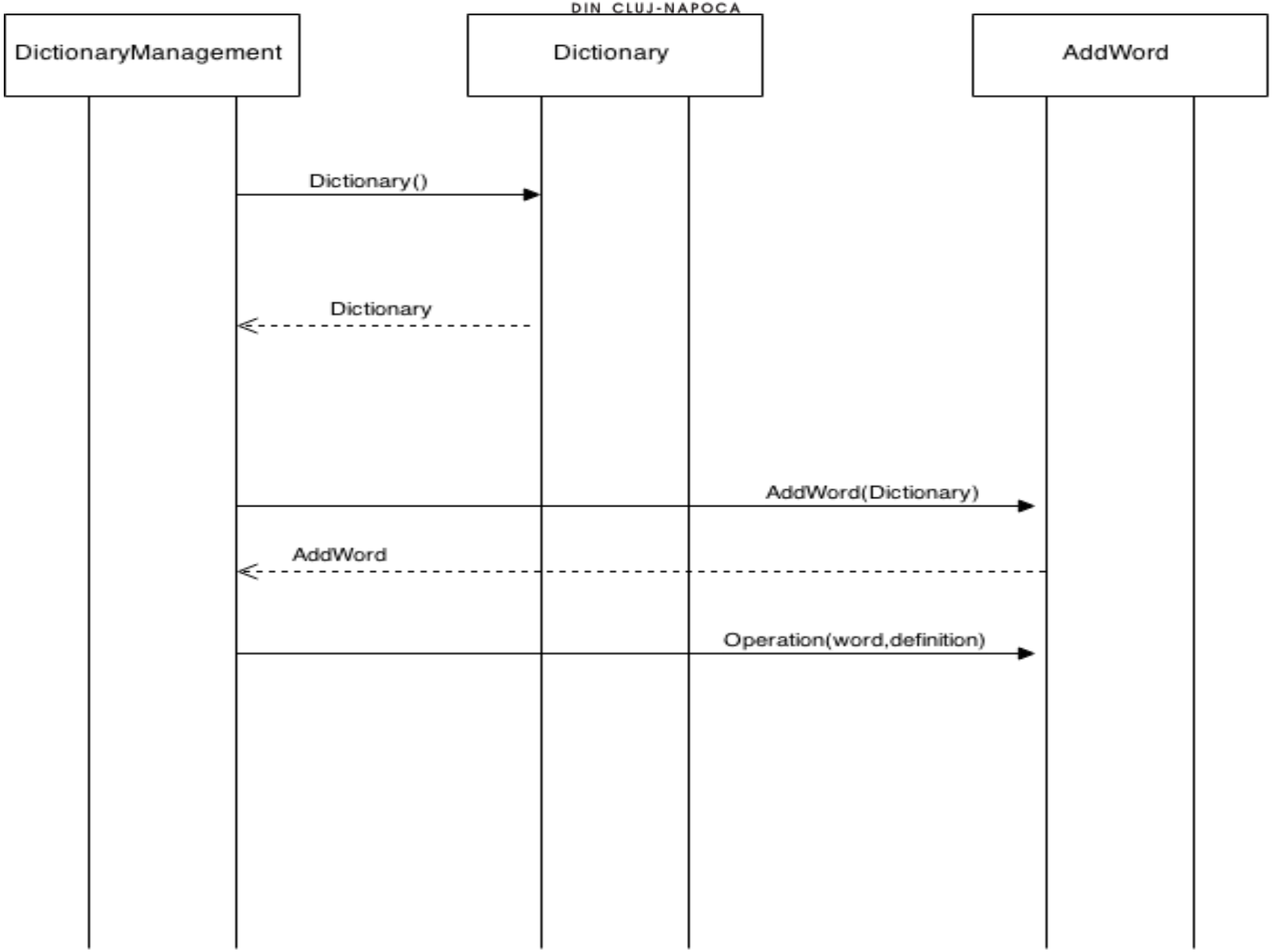
Șabloanele de proiectare reprezintă soluții generice la probleme des întâlnite în elaborarea aplicațiilor software. Utilizarea șabloanelor de proiectare este benefica din următoarele motive: . Soluțiile prezentate in cadrul șabloanelor de proiectare au fost analizate de întreaga comunitate a specialiștilor in programarea orientata-obiect pentru o perioada lunga de timp. Astfel, au fost studiate toate implicațiile utilizării unui anumit șablon, iar soluțiile preconizate au un nivel ridicat de calitate. . Șabloanele de proiectare reprezintă soluții care trebuie doar adaptate la necesitățile problemei pe care dorim sa o rezolvam. Timpul de dezvoltare al aplicațiilor care folosesc șabloane de proiectare este in general mai redus fata de aplicatiile care sunt dezvoltate pornind de la zero. Utilizarea șabloanelor de proiectare nu implică numai avantaje, în anumite situații pot apărea, de asemenea, si dezavantaje. De exemplu, pentru anumite aplicații, soluțiile oferite de șabloanele de proiectare pot sa fie mai complexe decât unele concepute special pentru aplicația respectiva. Aceasta complexitate este datorata faptului ca șabloanele de proiectare reprezintă soluții generale care trebuie sa poată fi aplicate la o clasa larga de probleme. De obicei, dezavantajele sunt relativ minore fata de beneficiile aduse de utilizarea șabloanelor de proiectare si pot fi minimizate prin utilizarea diferitelor specializări ale șabloanelor. Pentru descrierea șabloanelor de proiectare se folosesc diagrame UML însoțite de explicații suplimentare referitoare la aplicabilitatea șablonului, rolul componentelor șablonului, avantajele si dezavantajele utilizării acestuia etc.

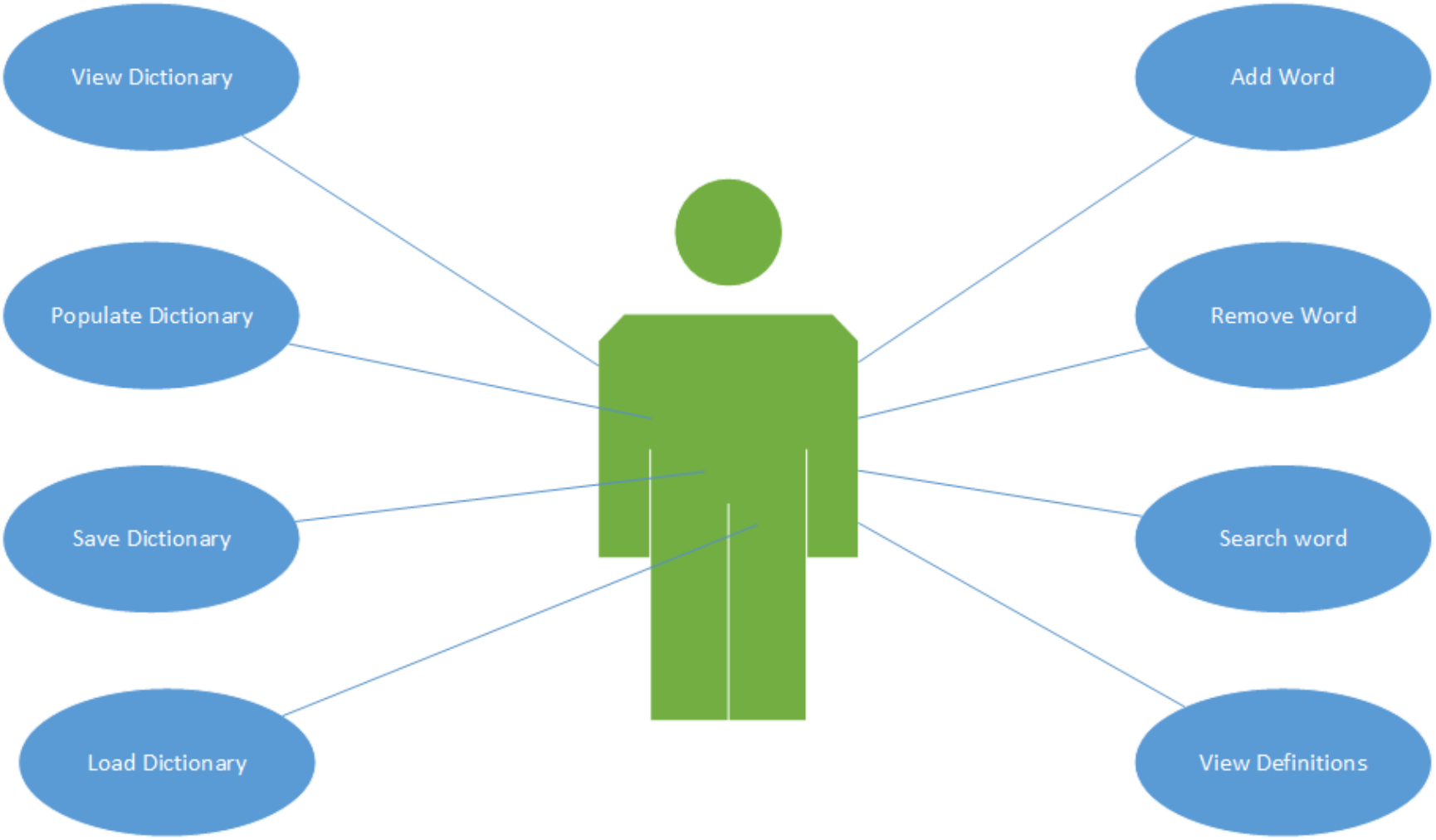
Șablonul decorator

Utilizare: adaugă caracteristici si elemente de comportament noi unui obiect in mod dinamic si transparent, fără a afecta alte obiecte. Poate fi utilizat pentru situații in care anumite responsabilitati ale obiectului pot fi acordate/ retrase. Reprezintă o alternativa la utilizarea subclaselor, mai ales atunci când folosirea acestora ar determina crearea unei structuri arborescente de clase si subclase foarte stufoasa.
5 In esența, șablonul Decorator reprezintă o soluție atunci când ne dorim sa cream un obiect, sa modificam comportamentul metodelor obiectului după ce l-am creat, sa revenim la vechiul comportament, totul fără a afecta codul client care interacționează cu obiectul.

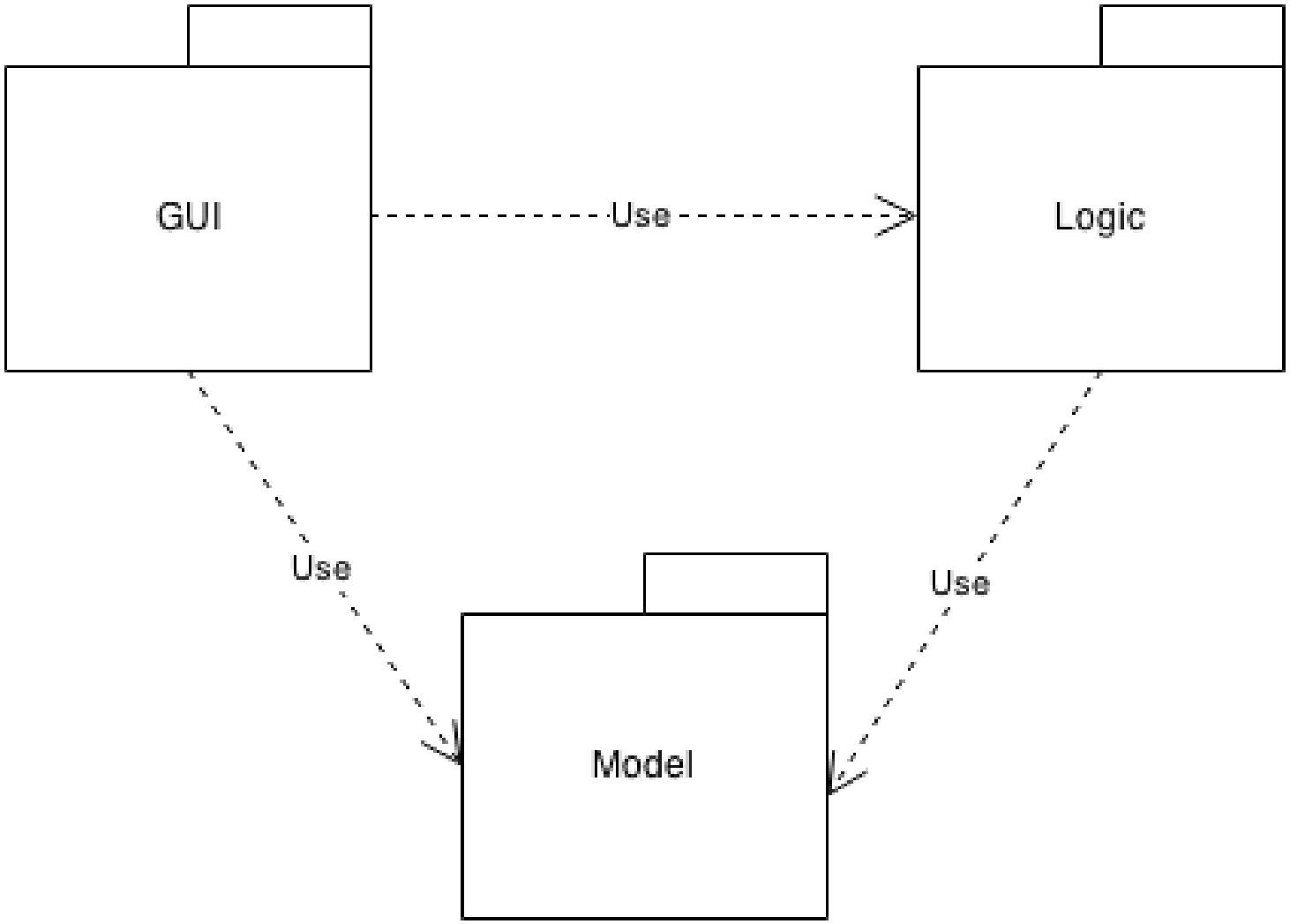
Sablonul Decorator (152) se ocupă de relatiile intre dase si obiecre, care suportă infrumusețare prin inchidere transparentă. Termenul ,,înfrumusețare" are de fapt un înteles mai larg decât cel prezentat de noi aici. In sablonul. Decorator, infrumusetarea referă"orice lucru care adaugi responsabilități la un obiect. Ca exemple de infrumusetare, ne putem gandi la un arbore abstract de sintaxă, cu acțiuni semantice, un automat cu stare finită cu tranziții noi sau o retea de obiecte persistente cu etichete atribut. Sablonul Decorator generalizează abordarea utilizari in programul DictionaryManagement pentru a o face aplicabilă pe scari mai largi.


Dictionary
- dic : TreeMap<String, String>
+ Dictionary()
+ Dictionary(TreeMap<String,String> inDic)
+ getDic() : TreeMap<String, String>
+ setDic(TreeMap<String,String> inMap)
Use
<<abstract>>
OperationDecorator
dic : Dictionary
+ OperationDecorator(Dictionary inDic)
# isWellFormed() : boolean
# Operation(String word, String description) : TreeMap<String,String>
Extends
AddWord
RemoveWord
WordMatcher
PopulateDictionary
SaveDictionary
LoadDictionary


DictionaryManagement
Dictionary
AddWord
Dictionary()
Dictionary
AddWord(Dictionary)
AddWord
Operation(word,definition)


View Dictionary
Populate Dictionary
Save Dictionary
Load Dictionary
Add Word
Remove Word
Search word
View Definitions

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA



4. Implementare și testare

Clasa Dictionary conține :

```
private TreeMap<String,String> dic;
```

Acest TreeMap<String, String> conține asocierea făcută între cuvânt și definiție. Menționăm că dicționarul este într-o stare consistentă în momentul în care fiecare cuvânt are o definiție în care cuvântul definit nu este utilizat. Acest lucru se face în metoda isWellFormed() din clasa OperationDecorator. Pe lângă această asociere, clasa Dictionary mai conține setterele și getterele necesare pentru a putea fi efectuate operații mai complexe asupra acesteia. Aceste settere și gettere urmează a fi decorate.

Clasa OperationDecorator conține :

```
protected Dictionary dic;
```

Aici se realizează legătura între dicționar și interfața care îl decorează.

Metoda isWellFormed() :

```
protected boolean isWellFormed()
	{
		for(Entry<String,String> entry : dic.getDic().entrySet())
		{
			if((entry.getValue().equals(""))||(entry.getValue()==null))
				return false;

	if(entry.getKey().toString().matches(entry.getValue().toString()))
				return false;
		}
		return true;
	}
```

Aceasta parcurge TreeMap-ul și dacă găsește inconsistențe, returnează false.

```
public abstract TreeMap<String, String> Operation(String word, String definition);
```

Metoda abstractă Operation este metoda prin care sunt decorate operațiile. Fiecare clasă care implementează o operație atât la nivel de dicționar, cât și la nivel de cuvânt va realiza concret

această metodă. Ea primește ca și parametrii String word și String definition care sunt fie ambii șiruri vide, fie ambii șiruri diferite de cel vid. În primul caz, se vor realiza operații pe dicționar, ceea ce nu au nevoie de anumite cuvinte introduse de utilizator. Al doilea caz realizează operații concrete pe cuvânt(adăugare, ștergere, căutare și orice alta operație gândită de cel care dorește să extindă clasa dată) de aceea cuvântul din asociere trebuie să conțină definiția sa. Corectitudinea operațiilor sunt asigurate de metoda isWellFormed() care dacă aceasta returnează o valoare falsă, înseamnă ca este violată consistența dicționarului și acesta este adus în starea anterioară care era consistentă. Dacă valoarea returnată este true, atunci înseamnă că totul este în regulă și se execute operația cu succes. Pentru partea de dezvoltare efectivă, toate testele se vor executa în JVM folosind opțiunea –ea care activează assert-urile, pentru a asigura corectitudine în parametrii dați.

5.Rezultate

În urma implementării unei aplicații menite pentru a stoca un dicționar de cuvinte al limbii engleze, s-a obținut un soft flexibil, ușor de extins și maleabil pentru nevoi atât personale, cât și muncitorești. Astfel, softul dat poate fi folosit atât în cadrul unei persoane fără specialitate în domeniul lingvistic, cât și de un specialist(ex :translator) și chiar și de un elev/student. Posibilitatea de a modifica dicționarul în funcție de nevoie permite schimbarea mediului de utilizare în domeniul științific prin realizarea unui dicționar științific, la nevoie.

6.Concluzii și dezvoltări ulterioare

În urma realizării softului dat, s-au repetat tehnicile de :

-serializare

- design by contract

și s-a învățat lucrul cu șabloane de proiectare.

Prin design by contract se stabilește un contract cu utilizatorul software-ului, astfel încât atât cel care a propus software-ul, cât și cel care îl utilizează trebuie să îl respecte. Daca oricare dintre cele două părți nu îl respectă, atunci contractul este încălcat și apare un bug în software.

Prin serializare se pot salva date într-un fișier sub formă binară, urmând ca mai apoi să fie deserializate pentru a fi citite(atât pe aceeași mașină, cât și pe alta)

Prin utilizarea șabloanelor de proiectare se construiește un soft flexibil, ușor de extins, performant ce îndeplinește condițiile tuturor producătorilor și ofertanților de pe piață.

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

7.Bibliografie

1. Cristian Frăsinaru – Curs practic de Java

2.Laborator Design după contract

3. Proiectarea orientată pe obiect din perspectiva ingineriei software - Luca Adrian

4. Curs serializare

5. Șabloane de proiectare : Elementele reutilizării software-ului orientat obiect - Banda celor 4