Parsed Use Case Descriptions as a Basis for Object-Oriented Class Model Generation

Mosa Elbendak, Paul Vickers and Nick Rossiter

Northumbria University
Newcastle upon Tyne, NE2 1XE, UK
<mosa.elbendak, paul.vickers>@northumbria.ac.uk

Abstract

Object-oriented analysis and design has become a major approach in the design of software systems. Recent developments in CASE tools provide help in documenting the analysis and design stages and in detecting incompleteness and inconsistency in analysis. However, these tools do not contribute to the initial and difficult stage of the analysis process of identifying the objects/classes, attributes and relationships used to model the problem domain. This paper presents a tool, Class-Gen, which can partially automate the identification of objects/classes from natural language requirement specifications for object identification. Use case descriptions (UCDs) provide the input to Class-Gen which parses and analyzes the text written in English. A parsed use case description (PUCD) is generated which is then used as the basis for the construction of an initial UML class model representing object classes and relationships identified in the requirements. PUCDs enable the extraction of nouns, verbs, adjectives and adverbs from traditional UCDs for the identification process. Finally Class-Gen allows the initial class model to be refined manually. Class-Gen has been evaluated against a collection of unseen requirements. The results of the evaluation are encouraging as they demonstrate the potential for such tools to assist with the software development process.

Keywords: Requirements specifications, object-oriented, parsing, analysis, natural language processing.

1. Introduction

The process of requirements identification is considered one of the most critical and difficult tasks in software design because most of the input to this process is in natural languages, such as English, which are inherently ambiguous. Developers need to interact with users in their own language. Also they need to review and analyze documents written in natural language. This paper introduces the Class-Gen CASE tool which was developed as part of an investigation into the problem of automatic identification of objects/classes and relationships from a requirements specification (RS) written in a natural language. Results from applying Class-Gen to some analysis problems are presented. First, existing literature on the subject of requirements specifications, object identification and conceptual database design is reviewed. Different techniques in natural language processing systems that attempt to transform natural language into conceptual models are considered for their effectiveness. Also examined are the rules to convert English sentences into entity relationship (ER) and enhanced entity relationship (EER) diagrams to determine entity types, attribute types and relationship types.

This work employs use case descriptions as the starting point for identification of classes and relationships and uses an existing parsing tool to identify noun phrases, verb phrases, adjectives and adverbs. This stage is based on existing work on how to map English sentences into conceptual models. The parsed use case description (PUCD) is proposed as an intermediate representation for capturing the output of the parsing stage, which is then used in subsequent steps. A PUCD is a set of original sentences, parsed sentences, nouns, verbs, adjectives and adverbs, which are used to extract nouns, verbs, adjectives and adverbs from use case descriptions. The next step is the process of identifying objects/classes, attributes, operations, associations, aggregations and inheritance so as to produce a class model. Classes are then refined by a human expert.

Following the literature review is an outline of a proposed method for object identification, which is based on existing work on how to map English sentences to conceptual models. Next comes a discussion of how to identify objects/classes, attributes, operations, and the association, aggregation and inheritance abstractions in order to produce class models by applying a set of rules dealing with correspondences between English sentence structures and ER modelling. The list of rules (see Appendix) was synthesised from several rule sets found in the literature and is implemented within the Class-Gen

system to enable it to produce class models.

2. Description of Problem

The successful development of any software system depends on the communication between clients and software developers to identify objects/classes and relationships from requirements specifications written in a natural language such as English, so as to increase efficiency in the use of scarce resources and to reduce errors in dealing with complex requests. Therefore, this work investigates how natural language processing tools and techniques can be used to support the object-oriented analysis (OOA) phase. The system works with English descriptions of the software requirements. For our purposes the requirements can be presented either as informal free text or in the more formalised format of a use case description (UCD). UCDs have been used in the proposed method as input for identifying classes and relationships as they are well-structured texts and are effective for analyzing and capturing functional requirements (for more details see Elbendak [22]). Automation of the systems development life cycle (SDLC) can help to alleviate critical problems of ambiguity, inconsistencies and conflicts in the functional requirements [34].

There are many problems and difficulties often associated with the use of natural language (NL) which can be summarised as follows:

- The ambiguity and complexity of NL are major problems in requirements specifications as they may lead to misunderstanding between the parties involved, which may badly affect customer satisfaction with the implementation produced. Furthermore, any errors, mistakes or inconsistencies incurred at this stage can be very costly later, especially when a system has already been implemented. It has been reported that the cost difference to correct an error in the early stage compared with leaving it till the end is 1:100 [8, 39, 37].

- The analysis process is considered to be one of the most critical and difficult tasks because most of the input to this process is in a natural language.

- Automatic identification of objects/classes and relationships is potentially faster than manual identifications but may be less accurate.

- There is no standard method for automatically identifying objects and classes from NL sentences.
- There is no standard template for requirement specifications. Therefore, the use case description may be more useful or effective in comparison to free form English prose as it is based upon functional requirements.
- There is little or no adoption of standards like UML for expressing requirements specification (RS) for the purpose of object identification.

Attempts to use NLP in the automatic generation of class models have been rather fitful, often being limited by the inherent difficulties in NLP (i.e. ambiguity and complexity, as mentioned above) and the last few years have witnessed a pause in development. Now two factors would seem to encourage a renewal of efforts in the area. First, dictionaries have become more sophisticated with the advance in ontologies, so that dynamic content targeted to a particular subject area can be used. Secondly, the capabilities of the NLP tools themselves have increased with respect to the syntactic classification of words in sentences (nouns, verbs, gerunds, etc). These two factors are of course linked in that the NLP tools themselves are also using more sophisticated dictionaries.

3. Background and Related Work

The identification of objects and classes is still often performed manually, using techniques such as textual analysis, common object lists, patterns and Class-Responsibility-Collaboration (CRC) cards. CRC cards, used as a manual technique in OO systems analysis methods, are considered to be an effective tool for conceptual modelling and detailed design [28]. There have been attempts to automate some of these manual approaches. For instance in the automation of CRC cards, Agile Modelling (AM) has been developed for the business architecture, domain object model and software design stage. AM provides tools for CRC cards to enable users to follow an active stakeholder participation practice. However to use the cards requires the specification to be decomposed into atomic statements, making it more appropriate for single requirements such as user stories, business rules, or system use cases [3]; this is not necessary in our semi-automated approach.

Previous studies have provided some rules for mapping natural language elements to object-oriented concepts. However, the coverage is incomplete. For example, Abbott [1] first suggested that nouns indicate classes and objects, while verbs can denote relationships. Researchers and software designers such as Booch [9] and Liang et al. [38] have concluded that object identification and the refinement process are ill-defined tasks because of the difficulty of heuristics and the lack of a unified methodology for the analysis and design stages. This is mainly due to the lack of a formalism for object-oriented analysis and design.

Although there are many projects focusing on CASE tools for object-oriented analysis and design, there are only a few focusing on the formalization and implementation of the methodology for the object model creation process. In addition they are not well developed for software design project that require collaborative working among members of a software design project team. Wahono and Far [53, 54] examined the issues associated with the methodology for collaborative object-oriented analysis and design with their OOExpert system.

Data Model Generator (DMG) is a rule-based design tool proposed by Tjoa and Berger [52] which maintains rules and heuristics in several knowledge bases and employs a parsing algorithm to access information from the grammar using a lexicon designed to meet the requirements of the tool. During the parsing phase the sentence is parsed by retrieving necessary information using the rules and heuristics to set up a relationship between linguistic and design knowledge. The DMG has to interact with the user if a word does not exist in the lexicon or if the input of the mapping rules is ambiguous. The linguistic structures are then transformed by heuristics into EER concepts. Although there is a conversion from natural language to EER models, the tool has not yet been developed into a practical system.

Gomez et al.'s ER generator [27] is another rule-based system which generates ER models from natural language specifications. The ER generator consists of two kinds of rules: specific rules linked to the semantics of some words in the sentences, and generic rules that identify entities and relationships on the basis of the logical form of the sentence and of the entities and relationships under construction. The knowledge representation structures are constructed by a natural language understanding (NLU) system which uses a semantic interpretation approach. Al-Safadi [2] presents a semi-automated approach for designing databases in enhanced ERD notation. In this approach, the natural language source is used for the semi-automated

generation of a conceptual data model. The work was concentrated primarily on the creation of a tool to convert a natural language description into a conceptual data model, in this case an enhanced ER model.

CM-Builder by Harmain and Gaizauskas [30] is a natural language based CASE tool which aims to support the analysis stage of software development in an object-oriented framework. The tool documents and produces initial UML conceptual models. The system uses discourse interpretation and frequency analysis in a linguistic analysis. For example, attachment of postmodifiers such as prepositional phrases and relative clauses is limited. Other limitations include the state of the knowledge bases which are static and not easily updateable nor adaptive. Bajwa et al. [5] have presented work in which the primary focus is upon the designing of a theory to analyze natural language texts fully and the subsequent development of a software tool UMLG (UML-Generator) based on an implementation of this theory. It is claimed that UMLG can extract requisite information from a natural language text and, thereafter, convert this into a UML class diagram. However, details of the system are not given and no evaluation of the method's performance has been published so its efficacy can not be determined at this time. Bajwa with Choudhary [4] has also put forward a rule based system that has the capacity to select from a natural language text the information required. The system understands the context and then extracts respective information. Meziane and Vadera [45] and Meziane [44] have implemented a system for the identification of VDM data types and simple operations from natural language software requirements. The system first generates an entity-relationship model (ERM) from the input text followed by VDM data types from the ERM.

Another tool for the measurement and validation of UML class diagrams is MOVA [16]. However, it was unable to identify OO constituents automatically. Thus, in order to identify classes, objects and their particular methods and attributes the system had to be supplemented through help from the user. Mich [46] and Mich and Garigliano [47] have described an NL-based prototype system, NL-OOPS, which is aimed at the generation of object-oriented analysis models from natural language specifications. This system has demonstrated how a large scale NLP system called LOLITA can be used to support the OO analysis stage.

Some researchers, also advocating NL-based systems, have tried to use a controlled subset of a natural language to write software specifications and build tools that can analyze these specifications to produce useful results.

Controlled natural languages are developed to limit the vocabulary, syntax and semantics of the input language. Macias and Pulman [40] discussed some possible applications of NLP techniques, using the CORE Language Engine, to support the activity of writing unambiguous formal specifications in English.

The research described above has provided valuable insights into how NLP can be used to support the analysis and design stages of software development. However, each of these approaches has limitations, which means that as yet NL-based CASE tools have not emerged into common use for OO analysis and design. Abbott [1], Booch [9], and Booch et al. [11] describe similar approaches but they have not produced working systems that implement their ideas. Meziane [44] and Meziane and Vadera [45] produced workable systems but these required an unacceptable level of user interaction such as accepting or rejecting noun phrases to be represented in the final model on a sentence by sentence basis as the requirements document is processed.

Giganto [25] and Giganto and Smith [26] have proposed a controlled language employed to write the requirements document and generated use case specifications. Their approach is aimed at obtaining classes from use cases, rather than directly from the specification. For example Giganto [25] proposed an algorithm to extract use case sentences from the requirements, validate the functional specifications of each sentence and reuse the validated domain dependent use cases to supply the missing functional specifications that may contain the participating classes. A limitation of this approach is that it is written in rules restricted to the functional requirements. By contrast, the advantage of the Class-Gen system (see below) is that it can be used by software engineers to generate a class model from the functional requirements. Mich and Garigliano's [47] approach, which is the closest to our proposed method, relies on the coverage of a very large scale knowledge base but the impact of (inevitable) gaps in this knowledge base on the ability of the system to generate usable class models remains unclear. It is also worth noting that none of these systems, so far as we are aware, has been evaluated on a set of previously unseen software requirements documents from a range of domains. This ought to become a mandatory methodological component of any research work in this area, as it has in other areas of language processing technology, such as the DARPA-sponsored Message Understanding Conferences (e.g., see Hirschman [32]).

Perez-Gonzalez and Kalita [48] have also proposed a semi-natural lan-

guage (4WL) to automatically generate object models from natural language text. Their prototype tool GOOAL [48] produces OO static and dynamic model views of the problem. Li et al. [37] also presented work to solve problems related to NL that can be addressed in OOA. Different NLP based tools have been proposed for this purpose. Zhou and Zhou [56] proposed another conceptual modelling system based on linguistic patterns. A framework was proposed to generate class diagrams from unstructured system requirement documents.

The methods and techniques discussed above are not automatic as they involve systems analysts taking many decisions during the OO analysis and modelling stage. On the other hand, these methods were dealing only with basic OO concepts.

Table 1 summarises the comparison above of existing object identification approaches with respect to techniques used, input types, breadth of coverage and automation.

Authors	Technique	Input	Coverage					Automated
			C/E	A	O	R	I	
Abbot [1]	Texual Analysis	RS in plain English	Yes	Yes	No	Yes	No	No
Chen [14, 15]	ER model	RS in plain English	Yes	Yes	No	Yes	Yes	No
Beck and Cunnigham [6]	Object identification	RS	Yes	Yes	Yes	No	No	No
Gomez [27]	Textual analysis	RS	Yes	Yes	Yes	No	Yes	semi-automatic
Wahono and Far [53]	Textual analysis	RS in UML use case	Yes	Yes	Yes	No	Yes	No
Mich and Garigliano [47]	Textual analysis	RS in plain English	Yes	Yes	No	No	No	No
Meziane and Vadera [45]	Logic Form Language	RS	Yes	Yes	No	Yes	Yes	semi-automatic
Harmain and Gaizauskas [30]	Object identification	RS	Yes	Yes	No	Yes	No	semi-automatic
Hartmann and Link [31]	Textual analysis	RS in plain English	Yes	Yes	No	Yes	Yes	No
Giganto [25]	Object identification	Use Case description	Yes	No	No	No	No	No
Dennis et al., [19]	Textual analysis	Use Case description	Yes	Yes	Yes	No	Yes	No
Al-Safadi [2]	ER model	RS	Yes	Yes	No	Yes	No	semi-automatic

Table 1: Comparison of Object Identification proposals
C/E: Class/Entity; **A:** Attribute; **O:** Operation; **R:** Relationship; **I:** Inheritance; **RS:** Requirement Specification

4. Strategy and Design

Class-Gen is a modular NL-based CASE tool which performs a domain independent OO analysis. Its input is a single software requirements document which it analyzes linguistically to build an integrated discourse model and extracts the main object classes and the static relationships among the objects of these classes. Class-Gen is written in Java (approximately six thousand lines of source code). The system produces two kinds of output: a list of candidate classes and a list of candidate relationships. The steps involved can be summarised as follows:

a) Take a set of functional requirements, use case descriptions or a problem statement description in natural language.

b) Use PUCD generation to analyze syntactically the informal requirements text and keep all the intermediate analysis results for further analysis.

c) Use the results produced by the PUCD generation to extract nouns, verbs, adjectives and adverbs from use case description as part of an identification process to identify objects/classes, attributes and the relationships among them.

d) Produce a class model by applying the set of rules given in the Appendix.

e) Produce a first-cut static structure model of the system from the extracted objects/classes in a standard format and use manual intervention to refine the initial class model.

5. Implementation

Figure 1 gives an overview of the identification of classes and relationships where UCDs represent the input to the process. Class-Gen is not an iterative tool. That is, it permits review by the user but is not iterative in its operation. As will be explained in more detail later, the parsing process as a whole involves a tokenizer as a preliminary stage, a sentence splitter, a part-of-speech tagger and chunking followed by the parser itself. The PUCD generated from this stage is a set of the original sentences, parsed sentences, nouns, verbs, adjectives and adverbs. A preliminary class model is generated from the PUCD, which is then refined by a human expert. The steps involved in deriving the class diagram from NL are listed below.

Step 1: Parse the use case description(s) using a memory-based shallow parser (MBSP) to generate noun and verb phrases.

Step 2: Generate the PUCD from the output of step 1.

Step 3: Identify the classes including the association, aggregation and inheritance abstractions from PUCD objects/classes using the system's rules to produce a class model.

Step 4: A human expert works with Class-Gen to refine the output of step 3.

5.1. Use case descriptions (UCDs)

UCDs written in a natural language are usually employed to specify functional requirements but their format is not standardized. If the requirements document is written in English no limitations are imposed on its form which can be in the structure of a general problem statement describing the software problem or a list of more detailed functional requirements. Therefore, because UCDs do not have a defined format or structure, Class-Gen will accept as input any plain text file containing the software requirements in English.

5.2. Comprehension of the input using memory-based shallow parsing

Memory-based shallow parsing (MBSP) is an essential component in text analysis systems for text mining applications such as information extraction and question answering [55]. Shallow parsing provides only a partial analysis of the syntactic structure of sentences as opposed to full-sentence parsing. Parsing includes the detection of the main constituents of the sentences (for example noun phrases (NPs) and verb phrases (VPs)). An MBSP system for English usually consists of the following modules:

Tokenizing: The tokenizer splits a plain text file into tokens. This includes, for example, separating words and punctuation, identifying numbers, and so on.

Sentence Splitting: The sentence splitter identifies sentence boundaries.

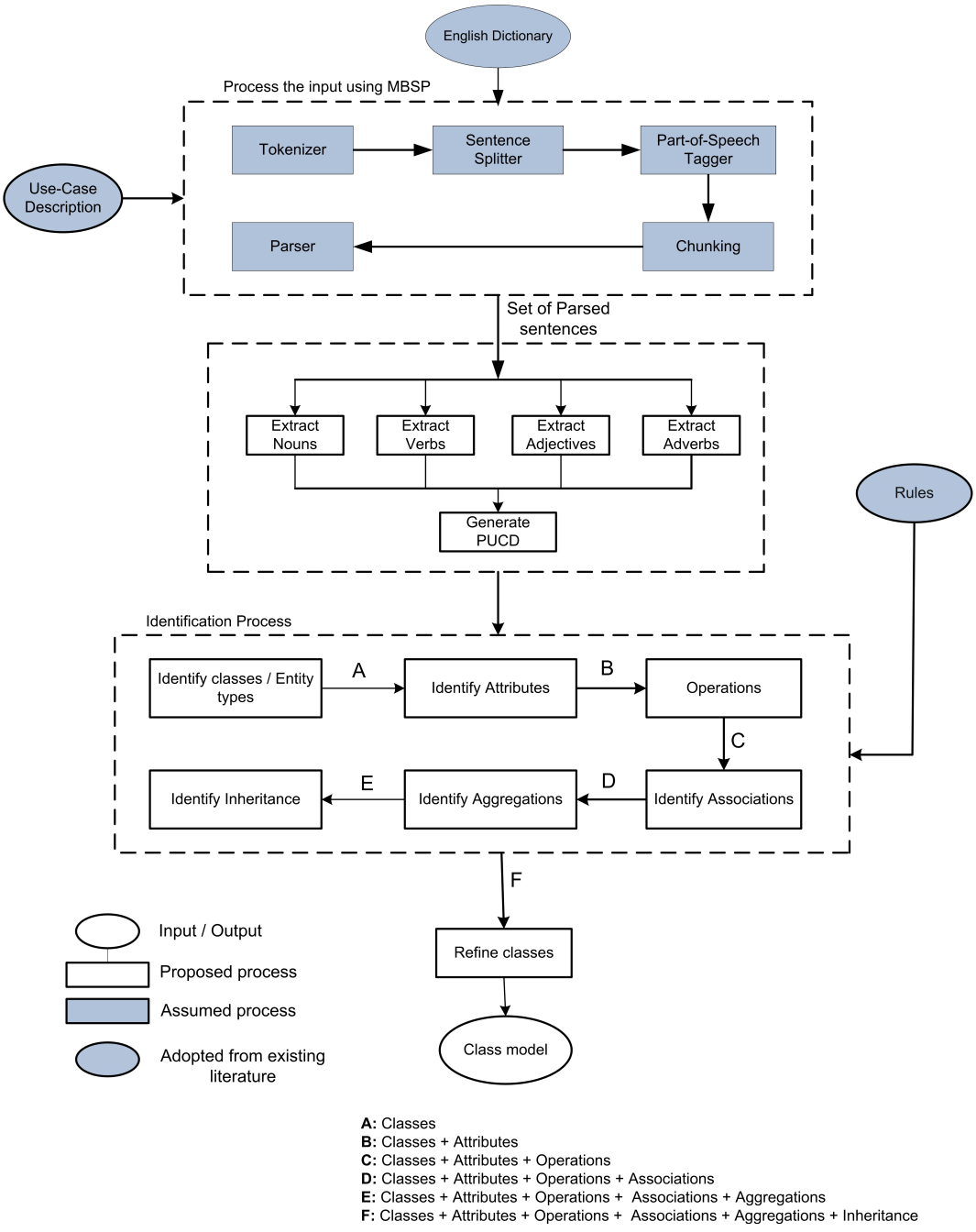

Figure 1: Overview of identification of classes and relationships

Part-of-Speech (POS) Tagging: The POS tagger assigns to each word in an input sentence its proper part of speech such as noun, verb and determiner to reflect the word's syntactic category. For example, for the clause *"The man likes the car"*, the POS tagger produces: *The/DT man/NN likes/VBZ the/DT car/NN* (see Brill [13, 12]).

Chunker: : Chunking is the process of detecting the boundaries between phrases (for example noun phrases) in sentences [18].]. Chunking can be regarded as light parsing. In MBSP, branded prediction is used during the task.

Parsing: Parsing is the process of determining the syntactic structure of a sentence given a formal description of the allowed structures in the language called a *Grammar*. An example of a parse tree for the sentence "The man likes the car" is shown in Figure 2.

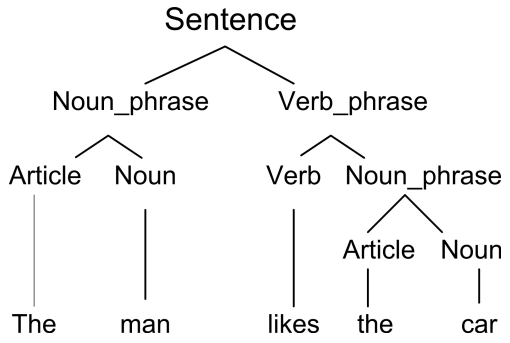

Figure 2: Parse tree for the sentence "The man likes the car"

5.3. Parsed use case description (PUCD)

The inputs to the PUCD generator are parsed and tagged as text. The main purpose of the PUCD generator is to extract the nouns, verbs, adjectives and adverbs so as to collect the class/entity type, attribute and relationship from the tagged input.

In some circumstances a single UCD may not be enough to provide all of the information required. Therefore, it is recommended to employ more than one UCD to cover all the information needed about properties such as attributes and relationships to produce an effective class model.

The parsed and tagged text PUCD is defined as a set of tuples as follows:

Ns	Nouns	Vs	Verbs	ADJs/ADVs	Adjectives/Adverbs
NN	Noun, singular or mass	VBD	Verb, past tense	JJ	Adjective
NNP	Proper noun, singular	VBG	Verb, gerund/ present participle	JJR	Adjective, comparative
NNPS	Proper noun, plural	VBN	Verb, past participle	JJS	Adjective, superlative
NNS	Noun, plural	VBP	Verb, non-third person singular, present	RB	Adverb
VB	Verb, base form	VBZ	Verb, third person singular, present	RBR	Adverb, comparative
RBS Adverb, superlative					

Table 2: List of PUCD Abbreviations

PUCD = {< OS, PS, Ns, Vs, ADJs, ADVs >}

Ns = {< N, tag>}, where N is any noun and tag ∈ {NN, NNP, NNPS, NNS}

Vs = {< V, tag>}, where V is any verb and tag ∈ {VB, VBD, VBG, VBN, VBZ, VBP}

ADJs = {< ADJ, tag>}, where ADJ is any adjective and tag ∈ {JJ, JJR, JJS}

ADVs = {< ADV, tag>}, where ADV is any adverb and tag ∈ {RP, RB, RBR, RBS}

OS is an original sentence

PS is a parsed sentence

Figure 3 shows an example of the PUCD generator extracting nouns, verbs, adjectives and adverbs from the requirement specification as original sentences, parsed sentences, Ns, Vs, ADjs and ADVs. Table 2 defines the abbreviations used in PUCD generation.

5.4. Object Identification Process

Details of the object identification process are given in Figure 1. After the extraction of nouns, verbs, adjectives and adverbs from the generated

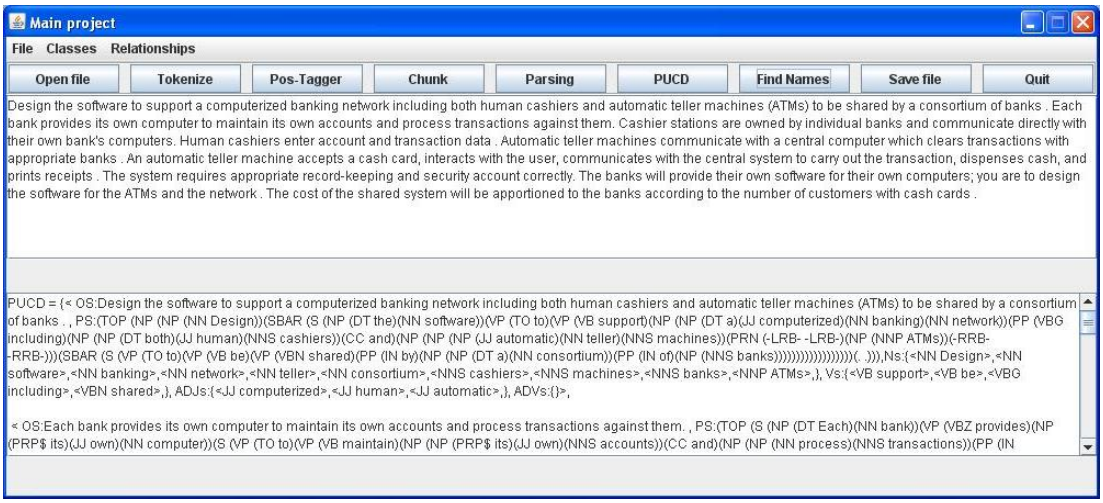

Figure 3: Example of the PUCD

PUCD, Class-Gen can then identify classes/entities, attributes and relationships using identification process rules.

5.4.1. Identifying Classes/Entities

Figure 1 shows the process for identifying classes/entity types, attributes, operations and relationships. Details for each step in the identification process are given below.

The first step is to produce a list of candidate classes. This is done by applying the noun identification rules (see Appendix) to the PUCD. A class is defined as follows:

$$C := \{< C_n, ATT, B, R >\}$$

where C_n is a class name, ATT is a set of attributes, B is a set of behaviours or operations and R is a set of relationships.

We identify a list of candidate classes and attributes as follows:

1. Determiners (a, an, the, each, and, with, etc.) do not play a crucial role at this stage, so they are ignored.
2. Plural noun phrases are converted to their singular form because class names in UML are given in the singular. For example, customers is changed to customer, and order items to order item.

3. Redundant candidates are removed from the list of the output PUCD. An exact string matching technique can be used to compare the candidates in the list with each other. For example, customer in our example sentences after the first instance is redundant, appearing many times in the text. The same is done for all other candidates.

Figure 4 shows an example of the list of candidate classes identified by the PUCD generator.

Figure 4: Example of the list of Candidate Classes

5.4.2. Identifying Attributes

A class C has a set of attributes $ATTs$ that describe the information for each object:

$$ATT := \{A | A :=< A_n, T >\}$$

where each attribute A has an attribute name A_n and a type T. The first step in the attribute identification process is to extract ADJ and ADV from the PUCD and then apply the attribute rules to examine the adjectives and adverbs in context to determine whether they are attributes of a class.

5.4.3. Identifying Relationships

There are four basic kinds of relationship: association, aggregation, composition and inheritance. Each class C has a set of relationships R. A relationship is represented by relationship type, related class and cardinality. A relationship R is defined as follows:

$$R := \{rel | rel :=< RelType, relC, Cr >\}$$

where $RelType$ is a relationship type (i.e., association, aggregation, composition or inheritance), $relC$ a related class and Cr a cardinality.

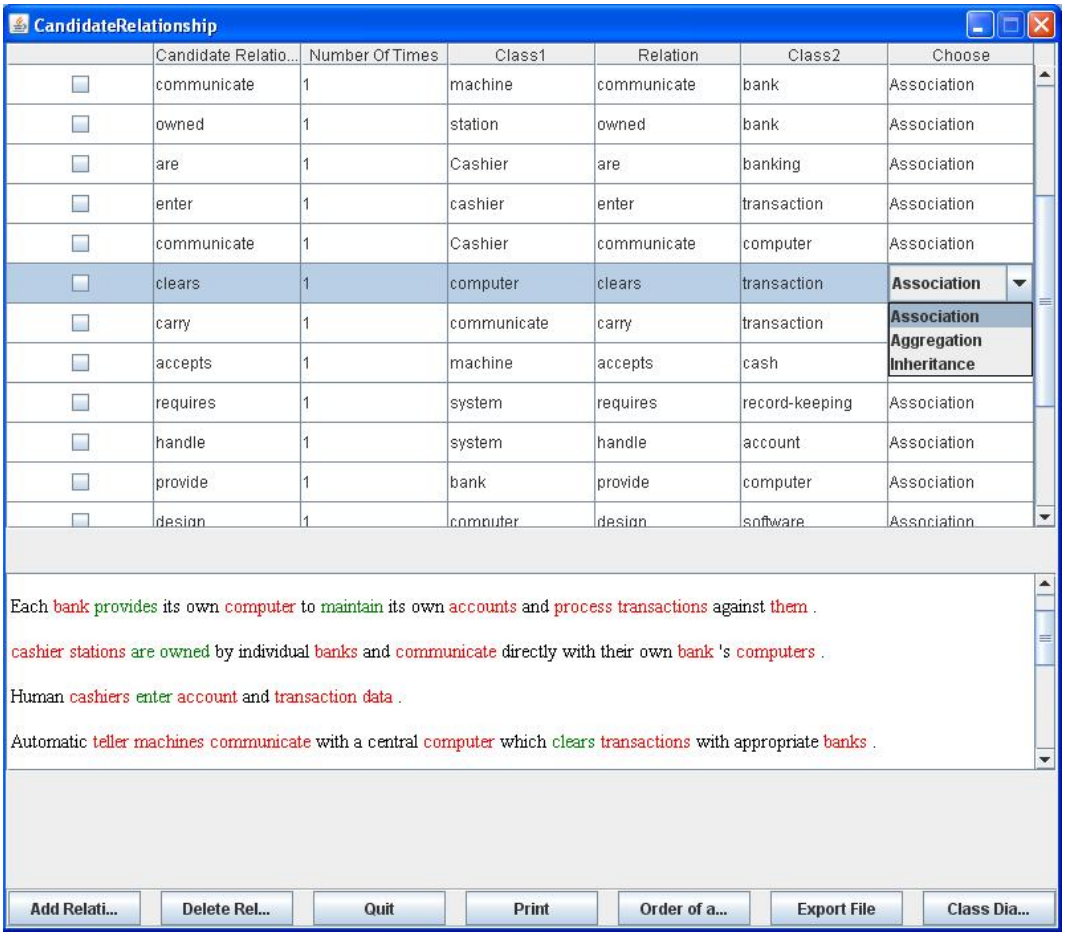

Figure 5: Example of the list of Candidate Relationships

Figure 5 shows an example of the list of candidate relationships identified from the PUCD generator.

Class-Gen identifies a list of candidate classes and candidate relationships from a PUCD by applying the rule set listed in the Appendix. Its processing steps are as follows (fully automated steps are prefixed with [A] whilst steps requiring some manual intervention have a prefix of [M]):

1. [**A**] Ignore articles, prepositions, connectives and other insignificant words (such as a, an, the, each, and, with, etc.) as they do not play a crucial role at this stage of identification.
2. [**A**] Convert plural noun phrases to their singular form because class names in UML are singular. For example, *Banks* is changed to *Bank*, and *Students* to *Student*.
3. [**A**] Identify all nouns in the text and treat each as a candidate class.
4. [**M**] Remove redundant candidates from the list of the output PUCD as they are not required. An exact string matching technique can be used to compare the candidates in the list with each other.
5. [**A**] Identify all verbs in the text and treat each as a candidate relationship.
6. [**M**] Determine the candidate class having the highest frequency of occurrence in the text (i.e., how many times it occurs).
7. [**A**] Consider all adjectives and adverbs in the text as candidate attributes.
8. [**M**] Add new words to the dictionary. Words or abbreviations occurring in the PUCD that are not known to the system are added along with their definition and part-of-speech type (noun, verb, etc.) to a dictionary so that they will be recognized next time they appear.
9. [**M**] Delete from the list all candidate classes having a low frequency and which do not participate in any relationship.
10. [**A**] Highlight nouns and verbs in different colours to facilitate the assignment of relationships between classes in the same sentence.
11. [**M**] Find aggregations by looking for clauses in the form "*something contains something", "something is part of something" and "something is made up of something".*
12. [**M**] Identification of roles in associations. Although this is not the main aim of the work the approach identifies three types of UML multiplicities [11]. e.g., *"1 for exactly one", "* for many" and "N for specific numbers".*

Applying the above steps leads to the production of an initial class model comprising the classes themselves, their attributes and relationships between classes.

6. Automatic Teller Machine Example

The Automatic Teller Machine (ATM) example presented here has been analysed by Rumbaugh et al. [50] using their OMT methodology. In this section we take the same problem statement used by Rumbaugh et al. and show the results of their analysis. After that we present the model produced by our system and compare it with Rumbaugh et al.'s model.

6.1. Problem Statement

The problem statement of the ATM as given in Rumbaugh et al. [50] is shown in Figure 6.

> *Design the software to support a computerized banking network including both human cashiers and automatic teller machines (ATMs) to be shared by a consortium of banks. Each bank provides its own computer to maintain its own accounts and process transactions against them. Cashier stations are owned by individual banks and communicate directly with their own bank's computers. Human cashiers enter account and transaction data. Automatic teller machines communicate with a central computer which clears transactions with appropriate banks. An automatic teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts. The system requires appropriate record-keeping and security provisions. The system must handle concurrent access to the same account correctly. The banks will provide their own software for their own computers; you are to design the software for the ATMs and the network. The cost of the shared system will be apportioned to the banks according to the number of customers with cash cards.*

Figure 6: The problem statement from Rumbaugh et al. [50, p. 151]

6.2. Rumbaugh et.al.'s Object Model

Rumbaugh et al. [50] used their Object Modelling Technique (OMT) methodology to build an object model of the ATM example. They started by building a list of candidate classes extracted from the problem statement

by considering all nouns. This list includes 23 candidates: *Software, Banking network, Cashier, ATM, Consortium, Bank, Bank Computer, Account, Transaction, Cashier station, Account data, Transaction data, Central computer, Cash Card, User, Cash, Receipt, System, Recordkeeping Provision, Security provision, Access, Cost,* and *Customer*.

After producing this list, they considered all verb phrases in the problem statement and built an initial object model that shows classes and associations as given in Figure 7.

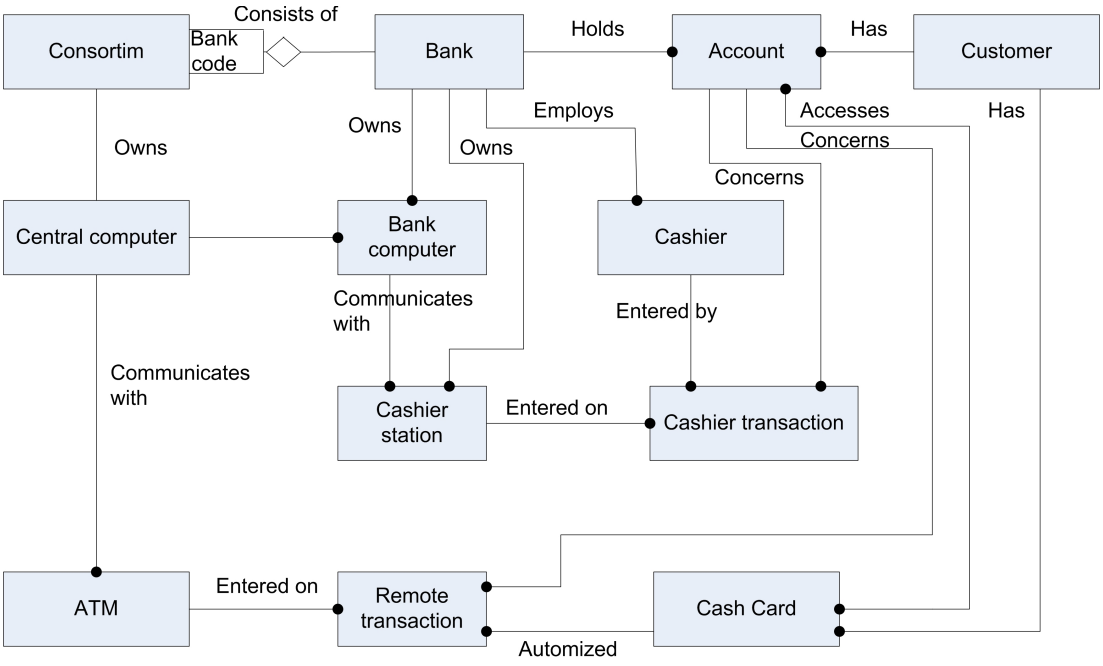

Figure 7: ATM initial object model from Rumbaugh et al. [50, p. 161]

6.3. Class-Gen Analysis

Classes and Relationships

From the ATM problem statement Class-Gen initially produced 55 candidate classes and attributes and 17 candidate relationships. The phrase frequency analysis process reduced the number of candidates to 27. The compound noun and attribute analysis process, further reduced the total number of candidates to 21. The candidate relationships generated from the

event nodes in the semantic net are also filtered by discarding any candidate that does not associate at least two classes. This process reduced the total number of candidate relationships to 8.

We now have a list of refined candidate classes with their attributes and a list of refined candidate relationships. After refinement we have produced a model of 11 classes and 8 associations as shown in Figure 8.

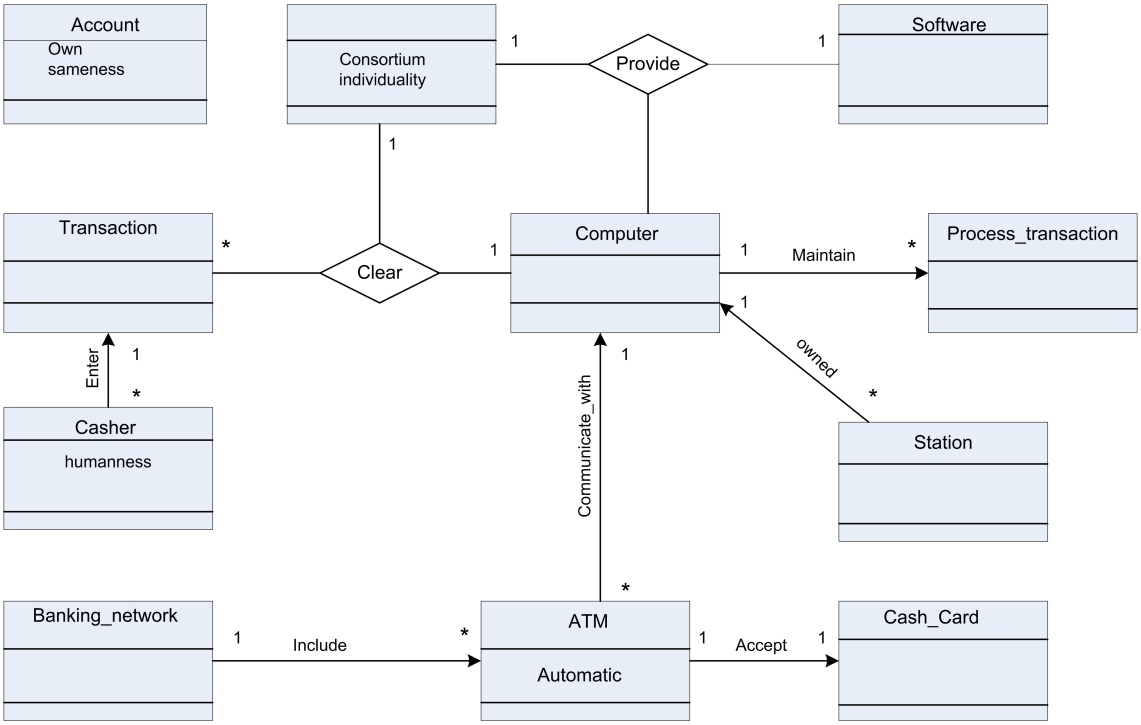

Figure 8: A Class Model of the ATM problem produced by Class-Gen

Comparison with Rumbaugh Model

Nine of the classes in first model produced by Class-Gen (Figure 8), are also shown on the model produced by Rumbaugh et al. (Figure 7). These classes are: *Account, Bank, Station, Cashier, Transaction, Cash-Card, Teller-Machine,* and *Computer*. The class *Transaction* is modelled by Rumbaugh et al. [50] as two classes: *Remote Transaction* and *Cashier Transaction*. Three of the classes identified by our system (*Software, Banking_network*, and *Access*) have been discarded by Rumbaugh et. al. on the basis of their vagueness, a concept which our system does not handle. The

associations identified by our system are not the same as those produced by Rumbaugh et. al., although 8 out of 9 associations are correct. The only remaining association is *owned* (as the station owns the computer). As this relationship has no corresponding action we judge it to be 'incorrect' rather than 'extra'.

7. Evaluation

In order to test Class-Gen's performance, evaluation was conducted by applying Class-Gen to several previously unseen software requirements documents. Evaluation plays an important role in software development both for developers and for consumers. Hirschman and Thompson [33] discuss three kinds of evaluation:

7.1. Adequacy Evaluation

Adequacy Evaluation refers to the determination of system fitness for some particular task. This kind of evaluation is usually used to answer questions such as: will the system do what is required? how well will it do it? what cost will be associated with it?, etc.

7.2. Diagnostic Evaluation

Diagnostic Evaluation is used by system developers to test their system during its development. A large amount of test data is required for this kind of evaluation. This data is used to determine the coverage of the system and to fix any flaws found.

7.3. Performance Evaluation

Performance Evaluation is a measurement of the system performance in some area of interest. Many concepts from quantitative performance evaluation in information retrieval have been imported into the development of evaluation methodologies in NLP. Hirschman and Thompson [33] address three important concepts of performance evaluation that must be taken into account in any evaluation methodology:

Criterion: what we are interested in evaluating (e.g. precision, speed, error rate, etc.). The criterion applied relates to how closely the models produced by analysts match those produced by our approach (system responses versus answers key) [33]. However, a single gold standard

model for any given software requirement does not exist, as different human analysts will usually produce different models. These models cannot be categorised as strictly correct or incorrect, but nonetheless they are usually categorised as good or bad, depending on the objects and the relationships represented within them. It was assumed in this work that the models available in object-oriented text books are good models and so we have used them as usable answer keys.

Measure: relates to the system performance obtained given the chosen criterion (e.g., ratio of hits to hits and misses, seconds to process, percent incorrect, etc.). We have used two metrics for evaluating our system, *recall* and *precision*, which were originally developed for evaluating information retrieval systems and are the basic measures used in evaluating search strategies. In any system, both precision and recall should ideally be as close to 100% as possible.

Recall: Recall reflects the completeness of the results produced by an information extraction (IE) system [29]. The correct and relevant information returned by the system is compared with that found in the answer key. Recall is defined as:

$$\text{Recall} = \frac{N_{correct}}{N_{correct} + N_{missing}}$$

where $N_{correct}$ refers to the number of correct responses made by the system, and $N_{missing}$ is the number of information elements extracted by human expert and missed by the system.

Precision: Precision reflects how much of the extracted information was correct. The following formula is used to calculate precision:

$$\text{Precision} = \frac{N_{correct}}{N_{correct} + N_{incorrect}}$$

where $N_{correct}$ is as above, and $N_{incorrect}$ refers to the incorrect responses made by the system [29].

Method: to determine the appropriate value for a given measure and a given system. A manual method has been used where the results produced by the system are compared with the answer key. The Class-Gen system determines the classes, attributes, associations, generalizations

and inheritances. Each of the correct answers that matches an element in the answer key is considered correct. If the result does not match an element in the key it is set as incorrect. It is set to *extra* if the answer is valid information from the text but it does not exist in the key answer.

Correct: an element in the answer model is said to be correct if it exactly matches (i.e. exact string match of names) an element in the key model. If no exact match exists, we use the problem statement and our own judgment to find any element in the key model which is semantically identical to the element of the system answer (i.e. both of them refer to the same entity).

Incorrect: If an element in the answer model is not in the key model it is said to be incorrect, and both the problem statement and our judgment confirm that it is wrong (i.e. if an adjective, adverb, or a verb in the problem statement is wrongly given as a class).

Extra: An element in the answer model is said to be an extra element if it is correct (again by judgment and using the definition of the element in UML), but is not in the key model. This should not be confused with the term 'spurious' in message understanding conference (MUC) conferences which refers to the system extra incorrect responses.

Missing: When an element is in the key model but not in the answer model it is considered as a missing element.

Over-specification: is an evaluation metric to measure how much extra information in the system answer is not in the key model generated by human analysts. The OOA community agreed that in the early stages of analysis it is recommended to over-specify rather than to miss important information [41, 35, 42, 43].

$$\text{Over-specification} = \frac{N_{extra}}{N_{correct} + N_{missing}}$$

Where N_{extra} is the number of extra elements retrieved and $N_{correct}$ and $N_{missing}$ are as previously defined.

8. Comparative Analysis

8.1. Raw performance measurement

A corpus of six case studies, each from a different domain and extracted from a text book, was used to measure Class-Gen's performance. This choice was dictated by the fact that the case studies are well known to software engineers having intermediate length and with solution available. None of these case studies was examined in detail prior to the final evaluation, nor was the system run on any of them before the evaluation. The requirements definitions in the case studies range from 100 to 550 words in length, with sentence length ranging between 5 and 39 words. The average sentence length was 18 words.

Table 3 depicts the scores of the Class-Gen system on the six case studies. Each row shows the scores for one case study and the last three columns show the recall, precision and over-specification of the Class-Gen system. From this it can be seen that the system's average recall, precision and over-specification were 90%, 85% and 45% respectively. Whilst recall and precision should be as high as possible over-specification should be as low as possible. The initial class list is close to that finally obtained because of good performance on over-specification.

As formal evaluation of other NL-based CASE tools has not been published in the literature we cannot compare our results with them. However, it is worth noting that other language processing technologies, such as information retrieval systems, information extraction systems and machine translation systems, have produced commercial applications with recall and precision figures well below the levels achieved by Class-Gen. The results of this initial performance evaluation are very encouraging and demonstrate the efficacy of the Class-Gen approach.

8.2. Comparison with human experts

Following the raw performance measurement further evaluation was conducted to compare the precision and recall obtained using Class-Gen with the performance of human analysts constructing a class model manually without the benefit of Class-Gen to assist them. Nine independent software engineers with between 5 and 20 years of experience in UML were asked to derive a class model from the requirements specification given in the UCD for the ATM case study used above. The subjects' performance was measured as above and the results are shown in Table 4.

Case Study	N_{cor}	N_{inc}	N_{mis}	N_{ext}	Recall %	Precision %	Over-specification %
ATM [50]	11	1	1	2	100	91	16
VSSR [49]	6	0	0	1	100	100	16
JRP [21]	5	1	0	2	100	83	40
EFP [20]	7	2	3	5	70	77	50
LHP [21]	4	1	1	3	80	80	60
BAMS [36]	8	2	1	6	88	80	88
Average	7	1	1	3	90	85	45

Table 3: Summary of evaluation results from all case studies
ATM: Automated Teller Machine; **VSSR:** Video Store; **JRP:** Journal Registration Problem; **EFP:** Electronic Filing Program; **LHP:** Local Hospital Problem; **BAMS:** Bank Accounts Management System.

The average performance of the nine subjects was 58% for recall and 82% for precision. From Table 3 we can see that Class-Gen's average performance of 90% and 85% respectively compares favourably. Class-Gen's performance on the ATM case study was even better than that obtained manually by the subjects, with recall and precision of 100% and 91% respectively.

Subject	Recall %	Precision %
1	72	66
2	90	90
3	36	80
4	54	85
5	54	85
6	72	100
7	45	85
8	45	50
9	54	100
Average	58	82

Table 4: Manual results of recall and precision obtained by nine software engineers

9. Conclusion and future work

This paper has proposed an approach to assist with the process of object identification. By extracting nouns, verbs, adjectives and adverbs from use case descriptions a new intermediate representation called parsed use case

descriptions (PUCDs) was introduced as a means of storing the results of NLP parsing stages.

The PUCDs were then incorporated into a new tool called Class-Gen. This tool uses natural language processing techniques to analyze software requirements. Class-Gen is used for the identification of object classes, their attributes and the relationships between them to produce an initial class model. This class model can then be refined by a human expert. It was found that Class-Gen performed very favourably on a number of exemplar case studies of varying size, and that it enabled an expert to achieve higher recall and precision values than those obtained by analysts using an entirely manual approach.

An important aspect of Class-Gen is that it can be used to generate first cut class models more quickly than could be achieved by hand. It took the nine software engineers between one to seven hours to produce a class model manually (these figures were obtained through self reporting). Compare this to Class-Gen which took between one and thirty minutes to create a class model. This is because the Class-Gen is highly efficient at analysing the problem statement text. The analysis is done by differentiating the text into classes and relationships, through highlighting them with different colours i.e. the colour red is for classes and the colour green is for relationships. This is shown in Figure 5, which is a screen shot of the implementation process.

Class-Gen was evaluated in two ways. First its raw performance was measured by producing class models from six exemplar case studies and calculating the precision and recall thus obtained. Second, a comparative analysis was conducted by comparing Class-Gen's performance against the results obtained by a number software engineers deriving class models manually.

The results achieved have demonstrated that the approach implemented in Class-Gen may be of assistance in the early stages of object-oriented analysis tasks. The results are very encouraging and several follow-up lines of enquiry are suggested:

- There is no standard format for UCDs. Performance of systems such as Class-Gen might be improved if UCDs were made more formal by restricting the style of language used. However, one particularly pleasing aspect of Class-Gen is that is *does* work very well with UCDs of varying formality; it is already quite robust against informal natural language, and this is what helps to set it apart from some previous systems.

- It was shown in this work how NL-based CASE tools can be quantitatively evaluated by appropriating the recall and precision measures from the information retrieval domain. Further studies will be able to explore in more detail how tools such as Class-Gen can be used to assist with analysis of the characteristics of attributes and relationships.

- To further investigate the aggregation and inheritance relationships which are more difficult to analyze automatically than relationships between classes.

10. Acknowledgments

Many thanks to the Cultural Affairs Department, Libyan People's Bureau, Paris, for their help and financial support.

References

[1] Russell J. Abbott. Program design by informal English descriptions. *Commun. ACM*, 26(11):882–894, 1983.

[2] Lilac A. E. Al-Safadi. Natural language processing for conceptual modeling. *International Journal of Digital Content Technology and its Applications*, 3(3):47–59, 2009.

[3] Scott Ambler. *CRC Modeling: Bridging the Communication Gap Between Developers and Users*. John Wiley & Sons, Inc., New York, NY, USA, 2005.

[4] Imran Sarwar Bajwa and M. Abbas Choudhary. A Rule Based System for Speech Language Context Understanding. *Journal of Dong Hua University (English Edition)*, 23(6):39–42, 2006.

[5] Imran Sarwar Bajwa, Ali Samad, and Shahzad Mumtaz. Object oriented software modeling using NLP based knowledge extraction. *European Journal of Scientific Research*, 35(1):22–33, 2009.

[6] Kent Beck and Ward Cunningham. A laboratory for teaching Object-Oriented thinking. In *OOPSLA '89: Conference proceedings on Object-Oriented programming systems, languages and applications*, pages 1–6, New York, NY, USA, 1989. ACM.

[7] Simon Bennett, Steve McRobb, and Ray Farmer. *Object-Oriented Systems Analysis And Design Using UML*. McGraw-Hill International, London, UK, 2002.

[8] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall Advances in Computing Science and Technology Series. Prentice Hall PTR, October 1983.

[9] Grady Booch. *Object-oriented Analysis and Design With Applications*. Addison-Wesley Longman Publishing, Inc., 1991.

[10] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Longman Publishing, Inc., 1999.

[11] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Object Technology Series. Addison-Wesley Professional, 2nd edition, 2005.

[12] Eric Brill. A simple rule-based part of speech tagger. In *Proceedings of the third conference on Applied natural language processing*, pages 152–155, Morristown, NJ, USA, 1992. Association for Computational Linguistics.

[13] Eric Brill. Some advances in transformation-based part of speech tagging. In *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 722–727, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[14] Peter Chen. Entity-relationship modeling: historical events, future trends, and lessons learned. In *Software Pioneers: Contributions to Software Engineering*, pages 296–310. Springer-Verlag, New York, NY, USA, 2002.

[15] Peter Pin-Shan Chen. English sentence and entity-relationship diagrams. *Information Sciences*, 29(2-3):127–149, 1983.

[16] Manuel Clavel, Marina Egea, and Viviane Torres Da Silva. The MOVA tool: A rewriting-based UML modeling, measuring, and validation tool. In *JISBD 07:12th Conference on Software Engineering and Databases*, 2007.

[17] Thomas M. Connolly and Carolyn E. Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management*. Addison-Wesley, 4th edition, 2005.

[18] Walter Daelemans, Antal van den Bosch, Jakub Zavrel, Jorn Veenstra, Sabine Buchholz, and Bertjan Busser. Rapid development of nlp modules with memory-based learning. In *Proceedings of ELSNET in Wonderland*, pages 105–113, 25–27 March 1998.

[19] Alan Dennis, Barbara Haley Wixom, and David Tegarden. *Systems Analysis and Design with UML Version 2.0: An Object-Oriented Approach*. John Wiley and Sons, 2nd edition, 2009.

[20] Kurt W. Derr. *Applying OMT: a practical step-by-step guide to using the object modeling technique*. SIGS Publications, Inc., New York, NY, USA, 1995.

[21] Daniel Duffy. *From Chaos to Classes: Object-Oriented Software Development in C++*. McGraw-Hill Companies, 1995.

[22] Mosa Elbendak. Framework for using a Natural Language Approach to Object Identification. In *Proceedings of the Student Research Workshop held in conjunction with The International Conference RANLP-09*, pages 23–28, Borovets, Bulgaria, 2009. Shoumen, Bulgaria.

[23] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, United States of America, 2005.

[24] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Pearson Education, 2007.

[25] Reynaldo Giganto. Generating class models through controlled requirements. In *NZCSRSC-08, New Zealand Computer Science Research Student Conference*. Christchurch, New Zealand, 2008.

[26] Reynaldo Giganto and Tony Smith. Derivation of classes from use cases automatically generated by a three-level sentence processing algorithm. In *ICONS '08: Proceedings of the Third International Conference on Systems*, pages 75–80, Washington, DC, USA, 2008.

[27] Fernando Gomez, Carlos Segami, and Carl Delaune. A system for the semiautomatic generation of E-R models from natural language specifications. *Data Knowl. Eng.*, 29(1):57–81, 1999.

[28] Kathleen Arnold Gray, Mark Guzdial, and Spencer Rugaber. Extending CRC cards into a complete design process. *SIGCSE Bull.*, 35(3):226–226, 2003.

[29] Ralph Grishman and Beth Sundheim. Message understanding conference-6: a brief history. In *Proceedings of the 16th conference on Computational linguistics, Morristown, NJ, USA*, pages 466–471. Association for Computational Linguistics, 1996.

[30] H. M. Harmain and R. Gaizauskas. CM-Builder: A natural language-based CASE tool for object-oriented analysis. *Automated Software Engineering*, 10(2):157–181, 2003.

[31] Sven Hartmann and Sebastian Link. English sentence structures and EER modeling. In *APCCM '07: Proceedings of the fourth Asia-Pacific conference on Conceptual Modelling*, pages 27–35, Darlinghurst, 2007. Australian Computer Society, Inc.

[32] L. Hirschman. The evolution of evaluation: Lessons from the message understanding conferences. *Computer Speech and Language*, 12(4):281–305, 1998.

[33] Lynette Hirschman and Henry S. Thompson. Overview of evaluation in speech and natural language processing. In Ronald A. Cole, Joseph Mariani, Hans Uszkoreit, Annie Zaenen, and Victor Zue, editors, *Survey of the state of the art in human language technology*. 1996.

[34] Arthur M. Langer. *Analysis and Design of Information Systems*. Springer, United Kingdom, London, 3rd edition, 2008.

[35] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, Upper Saddle River, NJ, 2nd edition, 2002.

[36] Timothy C. Lethbridge and Robert Laganiëre. *Object-Oriented Software Engineering: Practical Software Development using UML and Java*. McGraw Hill, 2nd edition, 2002.

[37] K. Li, R. G. Dewar, and R. J. Pooley. Object-oriented analysis using natural language processing. In *Proceedings of 8th International Conference for Young Computer Scientists (ICYCS'2005)*, 2005.

[38] Ying Liang, Daune West, and Frank A. Stowel. An approach to object identification, selection and specification in object-oriented analysis. *Inf. Syst. J.*, 8(2):163–180, 1998.

[39] Dong Liu, Kalaivani Subramaniam, Armin Eberlein, and Behrouz H. Far. Natural language requirements analysis and class model generation using UCDA. In *IEA/AIE 2004: Proceedings of the 17th international conference on Innovations in applied artificial intelligence*, pages 295–304. Springer Verlag Inc, 2004.

[40] Benjamin Macias and Stephen G. Pulman. A method for controlling the production of specifications in natural language. *Comput. J.*, 38(4):310–318, 1995.

[41] James Martin and James J. Odell. *Object-Oriented Methods*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1995.

[42] James Martin and James J. Odell. *Object-Oriented Methods: A Foundation*. Prentice Hall PTR, United States of American, 1995.

[43] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall Professional Technical Reference, United States of American, 2000.

[44] Farid Meziane. *From English to Formal Specifications*. PhD thesis, University of Salford, UK, 1994.

[45] Farid Meziane and Sunil Vadera. Obtaining E-R diagrams semi-automatically from natural language specifications. In *Sixth International Conference on Enterprise Information Systems (ICEIS 2004)*, pages 638–642, Universidade Portucalense, Porto, Portugal, 14–17 April 2004.

[46] L. Mich. NL-OOPS: From natural language to object oriented requirements using the natural language processing system LOLITA. *Nat. Lang. Eng.*, 2(2):161–187, 1996.

[47] Luisa Mich and Roberto Garigliano. Nl-oops: A requirements analysis tool based on natural language processing. In *Proceedings of Third International Conference on Data Mining Methods and Databases for Engineering*, pages 321–330, Southampton, UK, 2002. WIT Press.

[48] Hector G. Perez-Gonzalez and Jugal K. Kalita. Automatically generating object models from natural language analysis. In *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 86–87, New York, NY, USA, 2002. ACM.

[49] Tom Rowlett. *The Object-Oriented Development Process: Developing and Managing a Robust Process for Object-Oriented Development*. Prentice Hall PTR, 2001.

[50] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Pearson Education, 1991.

[51] Veda C. Storey and Robert C. Goldstein. A methodology for creating user views in database design. *ACM Trans. Database Syst.*, 13(3):305–338, 1988.

[52] A. Min Tjoa and Linda Berger. Transformation of requirement specifications expressed in natural language into an EER model. In *Entity-Relationship Approach — ER '93: 12th International Conference on the Entity-Relationship Approach Arlington, Texas, USA, December 15–17, 1993*, pages 206–217. Springer Berlin/Heidelberg, 1994.

[53] Romi S. Wahono and Behrouz H. Far. OOExpert: Distributed Expert System for Automatic Object-oriented Software Design. In *Proceedings of the 13th Annual Conference of Japanese Society for Artificial Intelligence*, pages 456–457, 1999.

[54] Romi S. Wahono and Behrouz H. Far. Hybrid reasoning architecture for solving object class identification problem in the OOExpert system. In *Proceedings of the 14th Annual Conference of Japanese Society for Artificial Intelligence*, 2000.

[55] Jakub Zavrel and Walter Daelemans. Feature-rich memory-based classification for shallow NLP and information extraction. In Jurgen

Franke, Gholamreza Nakhaeizadeh, and Ingrid Renz, editors, *Text Mining: Theoretical aspects and applicationspplications*, LNCS, pages 33–54. Springer Physica-Verlag, 2003.

[56] Nan Zhou and Xiaohua Zhou. *Automatic Acquisition of Linguistic Patterns for Conceptual Modeling*. Drexel University, 2004.

Appendix A. Rules Used by Class-Gen

This section presents a set of rules that state correspondences between English sentence structures and ER modelling. These rules apply natural language processing in extracting knowledge from the requirements specifications. Below are specific rules for determining entity types, attribute types and relationship types.

A.1. Rules to determine entity types

Rules for determining entity types are mainly based on the studies of Chen [15] and Tjoa and Berger [52].

Case 1: All nouns (both common and proper) are converted to entity types [52]. This includes all types of nouns such as collective nouns, common nouns, count nouns, mass nouns. 'London' is an example of a proper noun, 'school' is an example of a common noun. For example:

> "A school has a principal and many teachers".

> "Mr. Tom Lewis works in the Human Resources department".

In the first example we notice that "school", "principal" and "teacher" are common nouns and therefore are considered as entity types/classes.

In the second example "Mr. Tom Lewis" and "Human Resources" are proper nouns denoting a specific person and place and therefore instances of two entity classes.

Case 2: An English gerund "corresponding to a relationship-converted entity type gerund may indicate an entity type which is relationship converted entity type" in an entity-relationship diagram [15, rule 9]. A gerund is a noun derived from a verb (also known as a verbal noun) and has the form: noun + ing.

> "Students may borrow many books and borrowing is processed by a member of the library staff".

The verb "to borrow" in this example appears as the gerund "borrowing" as the subject of the second clause. The verb "to borrow" corresponds to a relationship type and this has been changed into an entity type "borrowing". In this case an entity type may inherit attributes of the relationship type. Hence in this example, the gerund "borrowing" may have attributes of the relationship type such as "copy number" and "borrow date".

Case 3: A clause may indicate a high-level entity type which encompasses or includes lower-level entity and relationship types [15, rule 10]. In English the clause is regarded as the main construction which consists of a subject and a predicate. A clause or subject clause can be built upon another clause. For example:

"The doctor decides which medicine to be given to each patient".

The clause "which medicine to be given to each patient" is a sub clause of the verb "decides". Both "medicine" and "patient" are entity types while "to be given" corresponds to a relationship type. The entire clause implies a high entity type called "Prescription".

A.2. Rules to determine attributes

The rules for determining attributes are as follows.

Case 1: A noun which takes the general form of TERM_SUFFIX such as noun_id, noun_no, noun_type or noun_number may indicate an attribute type [51]; the TERM_SUFFIX representation is often used in database problem specifications. For example:

"Each textbook has a book_id and a title".

The noun "book_id" in the above example may indicate that it is an attribute type.

Case 2: A noun phrase which follows the phrase "identified by" may indicate the presence of attribute type [27].

"A person identified by person-id and surname, can own any number of vehicles".

"Suppliers are identified by supplier-id".

Case 3: An intransitive verb may denote an attribute type [15, rule 2]. An intransitive verb is a verb which does not need any element to complete its meaning. For example:

"The train arrives every morning at approximately 8.15 am".

In this example the verb "arrives" is intransitive as it does not take an object. The attribute lies in the "arrival time" of "8.15am". In other words, by mentioning the arrival time of a train an attribute is deduced.

Case 4: An adjective corresponds to an attribute of an entity type [15, 52]. An adjective is a word that describes a noun or a pronoun. An adjective adds some attributes to a noun or pronoun to make it specific. For example:

"The large photo album has extra charges on delivery".

The adjective "large" may indicate an attribute "size" of the photo.

Case 5: The genitive case is used when referring to a relationship of possessor or source by using the "of" construction (or using the possessive apostrophe). This case suggests an attribute function. This is clearly shown in the following example:

"The employee's name is stored".

The noun "name" may be an attribute of the entity type "employee".

A.3. Rules to determine relationship types

Relationship types can be determined using the following rules.

Case 1: An adverb can indicate an attribute of a relationship type [15, rule 4]. An adverb is a word or a phrase that describes or modifies any parts of language apart from nouns (which are modified by adjectives). It may refer to time, place, degree, manner, cause and circumstances. For example:

"The employee visits the site a maximum of twice a week".

The verb "visits" corresponds to the relationship type, nouns such as "employee" and "site" are considered as entity types. The adverbial phrase "twice a week" describes the frequency of visits. So an attribute called "frequency of visits" is linked to the relationship "visits".

Case 2: A transitive verb can be a candidate for a relationship type [15, rule 2]. A transitive verb is a verb which needs an object to complete the meaning of the sentence. For example:

"A borrower may borrow many books".

In this example the nouns "borrower" and "books" are entity types. "Borrower" is the subject and "books" is the direct object. The verb "borrow" is a transitive verb and it corresponds to a relationship type.

Case 3: If a sentence has the form "The X of Y is Z" we may treat X as a relationship between Y and Z. In this case, both Y and Z represent entity types [15, rule 6]. For example:

"The father of Ali Hasin is Nory Hasin".

Both "Ali Hasin" and "Nory Hasin" are proper nouns so we can say that they are both instances of particular entity types and that "father" is a relationship between them. If we assume that both "Ali Hasin" and Nory Hasin" refer to instances of the "person" entity type, we may say that "father" is a recursive relationship from "person" to "person".

A.4. Rule for determining cardinality types

Tjoa and Berger [52] presented a rule for determining cardinality types. Cardinality may be determined as follows.

Case 1: A noun or a prepositional phrase whose occurrence is singular gets a minimal and maximum cardinality of 1 [52]. For example:

"The single room is meant for only one guest".

In this example, "room" and "guest" are singular nouns and these may suggest that the cardinality is of type one-to-one.

A.5. Rule to determine Enhanced Entity Relationships (EER/UML)

The basic concepts of ER modeling are not sufficient to represent the requirements of the newer, more complex applications. The enhanced entity relationship (EER) model is a conceptual data model capable of describing the data requirements for a new information system in a direct and easy to understand graphical notion [24, ch. 4, 7][17, ch. 12]. Data requirements for a database and conceptual schema using EER schema are comparable to UML.

The EER model includes all concepts of the original ER model together with concepts of specialization/generalization and categorization associated with the related concepts of entity types described as superclasses and subclasses and the process of attribute inheritance. This section presents rules for the EER/UML concepts, association and aggregation, composition, and inheritance.

a) Associations

Associations usually represent bi-directional relationships between instances of classes. Bi-directional means objects of the associated classes are aware of each other. If an association is uni-directional a solid triangle can be shown next to the association name to show in which direction the association can be read [10].

In a requirements document associations are usually denoted by verb phrases. For example, the clause "students borrow books" denotes a *borrow* association between objects of the two classes *Student* and *Book*. Figure A.9 shows this association.

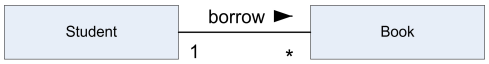

Figure A.9: Association

In the same way that there are instance of classes, there are also instances of associations. A link relates a pair of objects. For example, an instance of the association *Borrow,* in figure A.9, relates two objects of type *Student* and *Book*.

Multiplicity

A multiplicity specifies the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given to roles within associations or parts within compositions. A multiplicity is a subset of the non-negative open integers and can be a single value (e.g. 1 for only one) or an integer range (e.g. 1..7). A single star (*) is also used to denote an unlimited non-negative integer range.

Figure A.9 shows a one-to-many association between a student and a book. It means one student can borrow many books, and a book (copy of) can be borrowed by one student at a time.

Roles

The roles played by classes involved in an association can be connected to that association. Role names are shown at each end of the association where the association connects to the participating class. Figure A.10 shows husband and wife roles played by members of the *Person* class in a *married to* association.

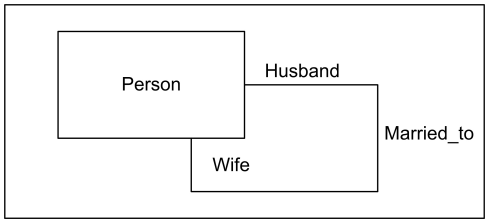

Figure A.10: UML Association Roles

Case 1: Association can be indicated when a clause has the meaning "B is associated-with A". This type of interpretation commonly shows an association between two entity types or an entity with its attributes. For example:

"The library has many book suppliers".

In this relationship, "book suppliers" may be regarded as an entity associated to "library".

"The book has a publisher".

In this case, "publisher" may become an attribute of "book" or another entity type linked to it. This depends on whether the existence of "publisher"

is highly significant in the business environment that is being analyzed in which other attributes of "publisher" (e.g. publisher's address, id and telephone number, etc.) are kept. It may also exist simply as an attribute of "book" (e.g. publisher's name).

Case 2 : A verb showing possession (e.g. "to have") may also imply an aggregation or association relationship [19]. For example:

"The car hire company has many branches".

In this type of "has/have" phrase, the noun that occurs after the phrase does not usually denote an attribute. The possession would normally show an association relationship between two entity types (classes). In the example above "car hire company" and "branches" are two distinct classes which are involved in an association relationship.

b) Aggregation

Aggregation is an important relationship between classes. Aggregation can be found by considering some clause patterns such as "*A is made up of B", "C is part of D", "E contains F"* [19].

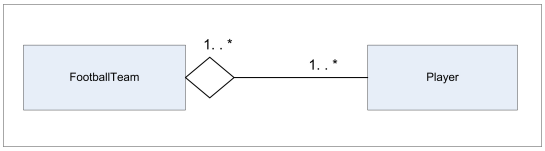

Figure A.11: UML Aggregation Relationship

Figure A.11 shows that a football team consists of 11 or more players. The aggregation relationship does not suggest a strong association between the parts and the whole. For example, a player can be a member of more than one team. In this case we can express the aggregation relationship as:

Aggregation = (*Player* is-part-of *Football team*).

c) Composition

Composition is a special kind of aggregation representing the existence of a strong relationship between the whole and its parts: a part cannot be a member of more than one whole. The multiplicity at the whole end is always one.

This means that, in a composite aggregation an object may be a part of only one composition at a time. For example, in a *BrickWall* system, a *Brick* belongs to exactly one *BrickWall*. This is in contrast to simple aggregation, in which a part may be shared by several wholes. For example, in the model of a house, a *Wall* may be a part of one or more *Room* objects. Figure A.12 shows that a brick wall is composed of one or more bricks.

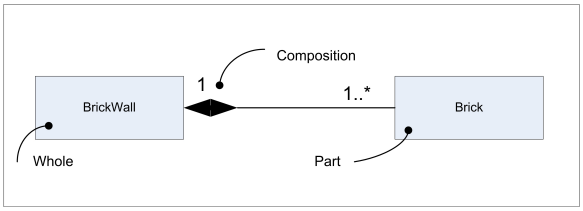

Figure A.12: Composition Relationship

d) Inheritance

Although it is not a new concept, having its roots in Artificial Intelligence (AI) knowledge representation paradigms, inheritance is a powerful concept in object orientation. The most important link in a semantic network is the '*is a*' relationship (is a type of) and such links support inheritance [10]. Thus, if X is a Y then X inherits all of Y's properties.

Suppose that a class, C is made up of set of attributes A and set of operations O.

$$C =< A, O >$$

Assume λ is a class. Let x, y and z also be the names of classes. Then:

$$\forall x \forall y [\lambda(x) \subset \lambda(y)]$$

and

$$\forall x \forall z [\lambda(x) \subset \lambda(z)]$$

Therefore,

$$< \forall y \forall z[\lambda(y) \cap \lambda(z)] > \; = \; < \forall x[\lambda(x)] >$$

is the property of inheritance where x is a superclass, and y and z are subclasses which inherit from x.

For example, consider the following class descriptions. The class *Student* has an attribute *name* and an operation *borrowbooks*; the *PGStudent* class has attributes *name* and *area of study* and operations *borrow books* and *borrow journals*; and the class *UGStudent* has attributes *name* and *stream* and an operation *borrow books*. We can represent these classes thus:

$$\begin{aligned} Student &= < A : \{name\}, O : \{borrowbooks\} > \\ PGStudent &= < A : \{name, areaofstudy\}, O : \{borrowbooks, borrowjournals\} > \\ UGStudent &= < A : \{name, stream\}, O : \{borrowbooks\} > \end{aligned}$$

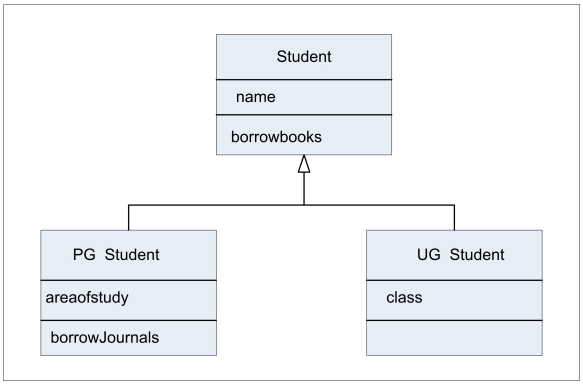

Figure A.13: Inheritance

Figure A.13 shows the relationships between these classes. The general, or superclass, *Student* is extended (inherited from) by the two specialized classes *PGStudent* and *UGStudent* (also called subclasses) representing postgraduate and undergraduate students respectively. The superclass *Student* defines both data and operations that are common to its subclasses. The subclasses inherit (reuse or share) the information defined in the superclass and add the information that makes them special.

Inheritance is considered an important concept in the object-oriented community for at least two reasons. First, it provides a very natural mechanism for organizing the information captured in OO models. Second it enables code and structure sharing, and hence assists software re-usability which is considered an important software engineering goal [23, 24]. Inheritance is evident in sentences that have '*is-a-kind-of*' relationships. For example, doctors, nurses and administrative personnel are all kinds of employees and those employees and patients are kinds of persons.

A.6. Rules to determine behaviour (Operation)

Operations define the way in which objects interact with other objects. They are used to model the services an object can perform. A behaviour model shows responses to events in the outside world and to the passage of time [7]. Behaviour is considerably more detailed when there is interaction between collaborating classes and when every message is defined as an event or an operation of a class; behaviour is identified by explicitly stating what the object does [10].

Operations are listed in the bottom compartment of a UML class. Figure A.14 shows that the *Student* class has two operations, *borrowBook* (b:Book) and *addCourse*(c:Course).

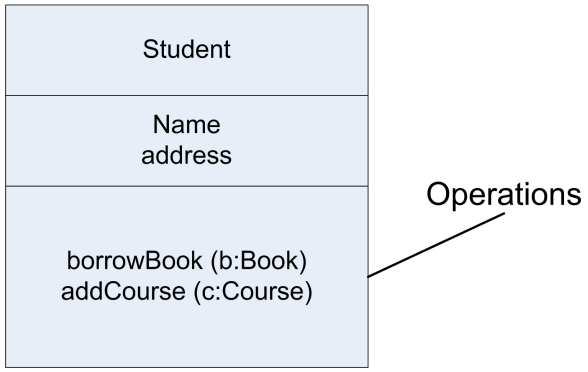

Figure A.14: Operation

Case 1: "A verb implies an operation" [19]. A verb is a word that shows action or state of being. For example:

"The author wrote many stories".

In this example, “wrote” is a verb indicating an operation.

Case 2: “A transitive verb implies an operation” [19]. A transitive is a verb that needs a direct object to complete its meaning. For example:

“Tom was crossing a bridge when the earthquake hit”.

In this example the transitive verb “crossing” indicates a possible operation.

Case 3: “A predicate or descriptive verb phrase implies an operation”. For example:

“Select the books on software engineering”.

In this example the predicate “select” indicates a retrieval operation.