Grammatically Driven Class Derivation (Extended Abstract)

Ernest Cachia

Software Engineering Process Improvement (SEPI) Research Group
Department of Computer Science and AI,
University of Malta

Abstract. This effort sets out to outline a research domain of academic and commercial relevance as well as the establishment of a possible research trend in the field of software engineering. The OO approach has established itself as a widespread and effective paradigm for modern software development. Many aspects of OO development are methodologically supported and procedural and representation standards are clearly defined. Certain activities within OO development remain suited for both automated and manual interpretations. It is also a fact that many system descriptions start off as natural language accounts of business processes, rather than semi-formalised data-flow or use-case models. It is therefore being proposed that a direct-from-text reliable and complete conversion method with governing standards can be defined to automate as necessary the class derivation activity, therefore decreasing the overall development effort and error-introduction probability without effecting objectivity within the OO development process. Such a conversion method would also allow more accurate rapid prototype generation at the earliest development stages. In theory, this would enable developers to automatically generate better quality "first-cut" GUI prototypes directly from textual system descriptions.

1 OO Development Effort

The effectiveness of engineering modern software systems has been an evolving multifaceted issue for a relatively long period of time [6] [7]. Several development paradigms exist each highlighting a "pathway" through the software engineering process [15]. An effective paradigm in modern software development is the Object-Oriented (OO) paradigm [4] [21]. The OO approach to system development is young enough to elicit research interest and yet, trodden enough to have proven its practical (commercial) aptness [11]. The advantages of OO-developed systems are discussed in numerous publications and are in particular elegantly outlined in Therekhov's paper [24]. Therekhov also sensibly justifies the effort for converting non-OO-developed ("legacy") systems into OO ones. The marked implications that the OO development paradigm has on software reusability and component-based software development, further highlight its potential.

2 OO Development Issues

One of the more essential requirements for effective OO system development is the adoption of a common development platform [20]. Such a platform should include models, procedures and guidelines. Without a common development platform, much effort would be misdirected causing many of the benefits which can be gleaned through the OO paradigm to be lost. In the course of the author's work experience within private companies supporting software development, it has been

repeatedly noted, that individual programmers exhibited definite but naturally specialised (hence limited) effectiveness. This situation is adequate for development of systems of pre-determined sophistication or the specific customisation of existing highly-sophisticated systems. However, once projects of considerable calibre are undertaken from inception, basic effort communication becomes crucial [16] [23] and the lack of a common development environment and standard modelling procedure considerably reduce development efficiency. This is an emerging phenomenon encountered by the author in many large and well-established companies in Malta.

3 OO Development Support with UML

One of the most prevalent development environments supporting the OO paradigm is the Unified Modelling Language (UML) [9] [18]. UML relies on a powerful combination of graphic and textual syntax to model various aspects of a system from an OO perspective. UML has given rise to several development processes closely based on it and is fast becoming an industry-standard for OO-based system development – mainly, but not solely, due to the continuous "fine-tuning" and enriching input of a host of prestigious commercial partner establishments as well as its ever-increasing user-base. One such development process currently enjoying widespread popularity is the Unified Software Development Process (USDP) or the Rational Unified Process (RUP), or simply the Unified Process (UP) [11]. USDP is made up of the following development phases [**?**]:

- Inception
- Elaboration
- Construction
- Transition

The phase where most of the design effort is concentrated is the Elaboration phase. This phase is made up of the following OO analysis steps:

- Class modelling
- Dynamic modelling
- Functional modelling

UML coverage of the Software Development Life Cycle (SDLC) ranges from the most abstract forms of system representation up to and including physical system models [3]. The research introduced through this document is mainly directed at issues in the Class modelling step.

4 Class Identification Issues

Although UML supports all the main phases in OO software development through precisely defined frameworks, procedures and guidelines, it nevertheless lacks any form of formal enforcement. This is particularly relevant to the traditionally more subjective activities in OO development. A typical example of such an activity would be the derivation of classes from a natural language description of a system [2]. This is the core activity of the class modelling step. Through the author's personal observation, it has been noted, that considerable system development errors are the result of incorrect or inappropriate derivation of classes from textual system descriptions. In the case of inexperienced developers or students, this form of error is manifested as missed, inconsequential

or wrongly derived classes. In the case of more experienced developers, the same form of error can be manifested as iterated ("multi-pass") class analysis leading to substantial development effort dissipation.

The most common approach adopted to try and limit the subjectivity of class identification is to offer developers a modelling tool that would allow system functionality to be specified at a high level of abstraction thus leading the developer to identify classes in full or partial fulfilment of the modelled functions. Such a modelling tool is the classical Data Flow Diagram (DFD) at context and first levels and later, the UML Use-Case Diagram (UCD) [19] [17]. However, DFDs at context level tend to be too abstract to offer any useful information regarding class identification, and DFDs taken to level one would have already required considerable effort and decision making in the first place. On the other hand, UCDs are often misinterpreted or trivialised based on the assumption that every use-case is realised by a class. In other words, the relationship between UML UCDs and UML Class Diagrams is not always straightforward and can be the source of development errors [25].

5 Prototype Generation Issues

Prototypes are one of the most helpful tools for both developers and users alike [10] [26]. Several valid approaches to generating prototypes exist [22] [27]. Most can be commonly summarised as follows:

1. Scope the system (with user input);
2. Determine the scenarios (with user input);
3. Model scenarios through UCDs;
4. Confirm scenarios (with user input);
5. Reconcile user viewpoints;
6. Produce prototypes.

Some fundamental issues can arise with the above generic (traditional) approach, namely:

- The user is really independently in the picture only while system description is in terms of natural language. Once descriptions move into the UCD domain, users will generally rely on developer interpretation of technical models to understand proposed system behaviour and then provide any corrective input. Users feel intimidated by this.
- Considerable effort is already in place before any form of tangible prototype is produced. This generally leads to predominance in the creation of "worth-it" prototypes.
- Prototypes created with this traditional approach tend to be biased towards whatever it is that the developer wishes (consciously or sub-consciously) to show.

6 Proposed Framework

The approach being proposed in this work is one that will allow users to feel comfortable in their system explanation and understanding and developers to feel comfortable and secure in their derivations.

The proposed technique is one that extends the basic traditional noun-verb analysis technique for class/method identification as pioneered by Russell J. Abbott [5]. The noun-verb class derivation

technique is only really effective when the relationship of classes to nouns is (or is close to) one-to-one [14]. The chosen grammar-based analysis of text is based on the breakdown presented by Dennis [8], which in the author's experience is one of the most comprehensive and credible. Dennis presents the following grammatical analysis mapping:

- ***A common or improper noun*** implies a class
- ***A proper noun or direct reference*** implies an object (instance of a class)
- ***A collective noun*** implies a class made up of groups of objects from another class
- ***An adjective*** implies an attribute
- ***A "doing" verb*** implies an operation
- ***A "being" verb*** implies a classification relationship between an object and its class
- ***A "having" verb*** implies an aggregation or association relationship
- ***A transitive verb*** implies an operation
- ***An intransitive verb*** implies an exception
- ***A predicate or descriptive verb phrase*** implies an operation
- ***An adverb*** implies an attribute of a relationship or an operation

It is well known that numerous lexical parsing techniques exist each having their own specialised application. In the case of extracting all or the key parts of speech from the above list, basic lexical parsing is all that is envisaged. The well established technique of parsing by chunks [1] is a possible adequate method. However, the choice, justifications and testing involved in the actual choice is a matter for future consideration and can indeed constitute a research topic in its own right.

One of the most comprehensive and well structured procedures for class/object and interface identification is presented by Lee [14]. It is being proposed, that given Dennis' grammar-based breakdown, Lee's identification procedures and some additionally derived rules, it would be possible to automatically generate direct (i.e. one-to-one noun-class assumption) class lists, proceed to optimise the direct class list, and even generate a first-draft class diagram, which is, in effect, the optimised classes with defined inter-relationships. It is also being proposed, that consideration be given to the possibility of generating first-draft GUI prototypes from class diagrams (attributes, operations and relationships). It can be argued, that having a fully specified class diagram (even a first attempt), enough data is present to infer upon the structure and behaviour of the corresponding GUI.

Figure 1 shows a generic overview of the class and prototype derivation technique being proposed.

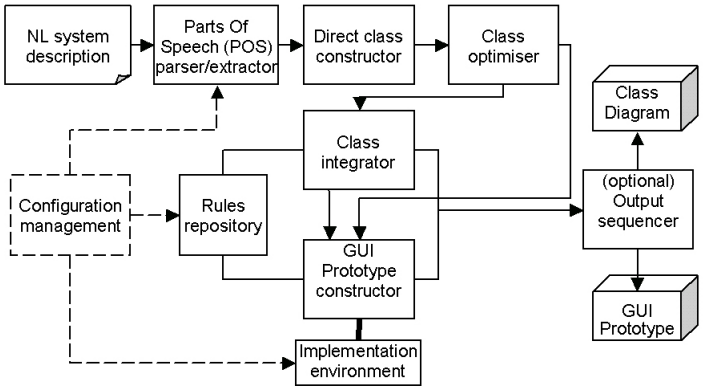

Fig. 1. Generic overview of class and prototype derivation technique

To conclude The effort outlined in this document can be classified as applicative and innovative. Applicative in the sense, that existing techniques are integrated and applied in a focused manner towards the betterment of the overall software development process. Innovative in the sense, that new interpretation and derivation methods can be properly researched and developed, which can eventually lead to levels of automation and objectivity so far unachieved.

The effort proposed is comprised of more than one possible research thread. One can clearly distinguish research threads in lexical analysis, data and behaviour derivation and correlation, interface inference, and possibly, notation (and semantic) development.

References

1. Abney, S. P., "Parsing in Chunks", Bell Communication Research, 1994.
2. Barnes, D. J., Kolling, M., "Objects First with Java", Prentice-Hall, 2003.
3. Bennett, S., et al., "Object-Oriented System Analysis and Design Using UML", 2^{nd} *Ed., McGraw-Hill,* 2002.
4. Booch, G., "Object-Oriented Analysis and Design with Applications", 2^{nd} *Ed., Benjamin-Cummings,* 1994.
5. Booch, G., "Object-Oriented Development", IEEE Transactions on Software Engineering, vol. 12, No. 2, pp.211-221, 1986.
6. Brooks, E. P., "The Mythical Man-Month: Essay on Software Engineering", Addison-Wesley, 1982.
7. Brooks, E. P., "The Silver Bullet, Essence and Accidents of Software Engineering", Kugler, H. J. Editor, Elsevier Science Publishers, 1986.
8. Dennis, "Object-Oriented Development", 2002.
9. Eriksson, H., Penker, M., "UML Toolkit", Wiley Computer Publishing, 1998.
10. Gilb, T., "Principles of Software Engineering Management", Addison-Wesley, 1988.
11. Jacobson, I., et al., "The Object Advantage: Business Process Reengineering with Object Technology", New York, ACM Press, 1995.
12. Jacobson, I., et al., "The Unified Software Development Process", Addison-Wesley, ACM Press, 1999.
13. Roff, J.T., "UML: A Beginner's Guide", McGraw-Hill, 2003.
14. Lee, R. C., Tepfenhart, W. M., "Practical Object-Oriented Development with UML and Java", Prentice-Hall, 2002.
15. Pfleeger, S. L., "Software Engineering Theory and Practice", 2^{nd} Ed., Prentice-Hall, 2001.
16. Pressman, R., Ince, D., "Software Engineering: A Practitioner's Approach", European Edition, 5^{th} Ed., McGraw-Hill, 2001.
17. Rosenberg, D., Scott, K., "Use Case Driven Object Modelling with UML: A Practical Approach", Addison-Wesley, 1999.
18. Rumbaugh, J., "The Unified Modelling Language Reference Manual", Addison-Wesley, ACM Press, 1999.
19. Rumbaugh, J., "Getting Started: Using Use Cases to Capture Requirements", Object-Oriented Programming Journal, Sept. 1994.
20. Sachs, P., "Transforming Work: Collaboration, Learning and Design", Communications of the ACM, vol. 38, No. 9, pp. 36-44, 1995.
21. Shlaer, S., Mellor, S., "Object-Oriented Systems Analysis: Modelling the World in Data", Prentice-Hall, 1988.
22. Shirogane, J., Yoshiaki, F., "GUI Prototype Generation by Merging Use-Cases", Waseda University, Japan, 2002.
23. Sommerville, I., "Software Engineering", 4^{th} Ed., Addison-Wesley, 1992.
24. Terekhov, A. A., "Automated Extraction of Classes from Legacy Systems", St. Petersburg State University, Russia, 2001.
25. `http://www.visual-paradigm.com`
26. `http://www.softwareprototypes.com`
27. Van Vliet, H., "Software Engineering: Principles and Practice", Wiley, 2002.