Current Issues in Software Engineering for Natural Language Processing

Jochen L. Leidner
School of Informatics, University of Edinburgh,
2 Buccleuch Place, Edinburgh EH8 9LW, Scotland, UK.
jochen.leidner@ed.ac.uk

Abstract

In Natural Language Processing (NLP), research results from software engineering and software technology have often been neglected.

This paper describes some factors that add complexity to the task of engineering reusable NLP systems (beyond conventional software systems). Current work in the area of design patterns and composition languages is described and claimed relevant for natural language processing. The benefits of NLP componentware and barriers to reuse are outlined, and the dichotomies "system versus experiment" and "toolkit versus framework" are discussed.

It is argued that in order to live up to its name language engineering must not neglect component quality and architectural evaluation when reporting new NLP research.

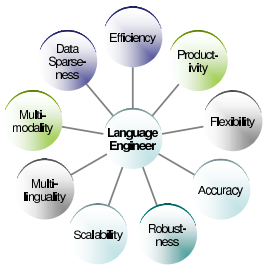

Figure 1: Dimensions of Language Engineering Complexity.

1 Introduction

It is notoriously difficult to construct conventional software systems systematically and timely (Sommerville, 2001), with up to 20% of industrial development projects failing. For Natural Language Processing (NLP) applications, the author is not aware of any studies that estimate project failure rate. The risks of failure seem even higher in this area, because the language engineer faces additional complexity (Figure 1):

Accuracy. A fundamental difference between NLP systems and conventional software is the *incompleteness property*: since current language processing techniques can never guarantee to provide all and only the correct results, the whole system design is affected by having to take this into account and providing appropriate fallbacks.

Efficiency. Human users are very demanding: (Shneiderman, 1997) reports that system response times $> 4s$ can render a system unacceptable. It is also debated in which scenarios natural language interaction with machines is superior to menus, keyboard commands or other means. To date, it unclear how efficiently NLP systems *can* be, since efficiency is not a top priority in research and many questions related to software design and technology are often considered a mere "implementation detail". This is in contrast to other areas of computing, where data structures and algorithms are often carefully selected and customized to be of maximum gain for a domain, and learning how to partition a problem into classes is seen as part of the knowledge acquisition process of attacking a problem.

Productivity. Time is a very scarce resource. Research environments often produce prototypes that demonstrate the feasibility of a method and leave efficient and more complete implementation to industrial exploitation. However, because in industry time is even more pressing, the re-implementation (from prototype to "production system") often doesn't happen. In research, productivity loss occurs because of lack of knowledge of exist-

ing resources, lack of trust in non-in-house components, or the inability to install or integrate existing software. Price or licensing concerns also play a role. It is argued here that software engineering techniques can improve overall productivity of researchers after some little initial investment. Section 3 relates short-term productivity to long-term gains to the choice between building a framework and carrying out an experiment.

Flexibility. Like any software, NLP systems need to be flexible: a parser developed primarily to analyze written online newspaper text might well be employed tomorrow to process business e-mails. Different data formats have to be handled, so representational and input/output knowledge needs to be factored out from core linguistic processing knowledge. Section 2.7 describes how design patterns can help NLP in this regard, and Section 2.2 gives an example of how componentization leads to more flexibility.

Robustness. In engineering, robustness refers to a device's ability to work even under varying conditions. In language engineering, the terms robustness and portability have obtained a more narrow meaning: both are typically used to describe the viability of a linguistic method when applied across different text types or domains (in terms of precision/recall).[1] But a decrease in robustness in the latter sense often means a "soft" degradation rather than complete system failure. This is the type of failure that needs to be handled even in a working system (→accuracy above), but decreases in overall performance are also more subtle and therefore difficult to detect.

Scalability. If an NLP system is to be deployed, it usually needs to be run in a distributed fashion to cope with a large number of users or documents. However, often complexity and observed runtime and memory results are not reported. In an engineering scenario, space and runtime limits are specified in advance by product management or lead architects and techniques are selected according to whether their properties are consistent with the requirements. Often NLP components rely on other components, and overall resource demands are high compared to the system the NLP component is part of (e.g. a text editor).

Multimodality. A language engineer applying the same parser to investigate the discourse structure of 18th century novels does not encounter the same challenges as her colleague trying to apply it to speech dialogs (e.g. absence of letter case information). Different modalities have their own idiosyncrasies, and it is difficult to factor out all of them, but this is necessary because there is a trend toward multi-modal systems, and intra-system reuse requires a high degree of adaptability.

Data Sparseness. Current NLP methods are often data-driven, which means they rely on the availability of a potentially large number of training examples to learn from. Such training corpora are typically expensive or virtually non-existent (data resource bottleneck). This holds even more so in a multilingual scenario. Insufficient training data yields unacceptable accuracy.

Multilinguality. In a globalized world, users want to work with a system in multiple languages. This is already an issue in conventional software: independence of character encodings, different lexicographic sorting orders, display of numbers, dates etc. (*internationalization*, or "I18N") need to be ensured and translations of iconic and textual messages into the user's currently preferred language of interaction (*localization*, or "L10N") have to be provided (Tuthill-Smallberg, 1997; Lunde, 1999). NLP applications are more complex, because grammars, lexicons, rule-sets, statistical models are language-specific, and need to be re-created for each new target language, often at a cost that is a significant proportion of the total NLP sub-system's budget. Often, heterogeneous components from different suppliers and following incompatible linguistic or software paradigms must be integrated. (Maynard et al., forthcoming) argue convincingly that architectural support can improve the predictability of the construction process.

2 Reuse

2.1 The need for reuse

In NLP, the global amount of reuse is low, and currently, activities of the community *en large* focus on reuse of data resources (via annotation standards or data repositories like LDC and ELRA). On the software side, despite similar efforts (Declerck et al., 2000), reuse rate is low, partially because the difficulty of integration is high (and often underestimated), for instance because developers use different implementation languages, deprecated environments or diverse paradigms. Especially, "*Far too often developers of language engineering components do not put enough effort in designing and defining the API.*" (Gambäck and Olsson, 2000). Thus, re-implementation and integration cause major productivity loss.

2.2 Properties that lead to reuse

How can productivity loss be avoided? Researchers should build their prototypes around sound Application Programming Interfaces (APIs); all input/output should be separated from the core functionality. Then not only will the workings of the algorithms become clearer, also the re-usability will be increased, since most applications make different assumptions about data formats. Potential sloppiness (e.g. lack of error-handling) caused by time pressure can then be restricted to the prototype application shell without impairing the core code. The main

[1] cf. the forthcoming *Special Issue on Natural Language Robustness* of the journal *Natural Language Engineering*.

LTG MUC-7	Hybrid MUC-7 Named Entity Recognizer based on maximum entropy classification and DFSTs
ltchunk	DFST-based English chunk parser
ltpos	HMM-based English POS tagger
ltstop	Maximum entropy-based English sentence splitter
lttok	DFST-based tokenizer for English text
LT TTT	Suite of XML/SGML-aware tools for building DFSTs
fsgmatch	Deterministic Finite-State Transducer (DFST) construction toolkit
sgdelmarkup	Remove SGML markup from text
sgtr	SGML replacement tool
sgsed	SGML stream editor
LT XML	LTG's XML API

Table 1: The SGML-Aware NLP Tools of the University of Edinburgh's Language Technology Group.

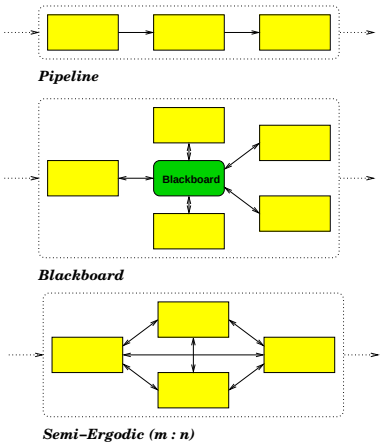

Figure 2: Some Global Architectural Choices.

principle behind good design is to dissect the problem domain into a set of *highly cohesive* components that interact in a *loosely coupled* fashion (Sommerville, 2001).

2.3 Barriers to reuse

Reuse of software components can be blocked by several factors, including the lack of knowledge of existing components, lack of trust in component quality, a mismatch between component properties and project requirements, unacceptable licensing policies or patent/cost issues. Political issues include the investment needed to make and package reusable components, for which there might not be any budget provided. Technical issues contain software-platform incompatibility and dependencies, installation difficulties, lack of documentation or support, and inconsistencies with other modules.

Considering NLP components in specific, formalisms might not be linguistically compatible. Components might differ in language coverage, accuracy and efficiency. With linguistic components, a black box integration is particularly tricky, since if the technique used internally is unknown, the component might break down in case the domain is changed (domain-specific rules/training). A further problem is posed by the fact that different paradigms perform sub-tasks on different levels (e.g. disambiguation). Case-sensitivity/case-awareness can also be problematic.

2.4 Code reuse: toolkits

The Edinburgh Language Technology Group's SGML-aware NLP tools (Mikheev et al., 1999) comprise a set of programs that rely on the common LT XML API[2] to annotate text using cascading (deterministic) Finite-State Transducers (Table 1).

The tools are typically used in a sequential UNIX pipeline (Figure 2, top). An integrated query language allows selective processing of parts of the XML/SGML document instance tree.

A major advantage of the LTG pipeline toolkit approach over frameworks (described below) is the maximal decoupling of its components (communication only by means data exchange in a "fat XML pipe"), so no toolkit-specific "glue" code needs to be developed and developers can work in their programming language of choice. A disadvantage is that repeated XML parsing between components may be too time-consuming in a production scenario.

2.5 Code and design reuse: frameworks

A framework is a collection of pre-defined services that embody a certain, given organization, within which the user can extend the functionality provided; frameworks impose certain organizational principles on the developer (Griffel, 1998).

The *General Architecture for Text Engineering (GATE)*[3] is a theory-neutral framework for the management and integration of NLP components and documents on which they operate (Cunningham et al., 1996; Cunningham, 2000; Bontcheva et al., 2002; Cunningham et al., 2002; Maynard et al., forthcoming). GATE 2 is compliant with the TIPSTER architecture (Grishman, 1995), contains the example IE system ANNIE and is freely available including source (in Java, which makes it also open for all languages due to the underlying use of UNICODE). A data type for annotating text spans is provided, which allows for generic visualization and editing components and a graphical plug-and-play development environment.

[2] http://www.ltg.ed.ac.uk/software/xml/

[3] http://gate.ac.uk/

Developers can make use of a sample component toolbox.

(Zajac et al., 1997) present Corelli, another TIPSTER compliant architecture implemented in Java (see (Basili et al., 1999) for a comparison). The WHITEBOARD project (Crysmann et al., 2002) uses monotonic XML annotation to integrate deep and shallow processing (Figure 2, middle). Finally, the closest coupling takes place in architectures where most or all components are allowed to talk to each other, such as the German Verbmobil speech translation system (Görz et al., 1996).

ALEP, the *Advanced Language Engineering Platform* (Simpkins and Groenendijk, 1994; Bredenkamp et al., 1997) is an early framework that focused on multilinguality. It offers an HPSG like, typed AVM-based unification formalism (and parsers for it) as well as some infrastructural support. In the LS-GRAM project, it has been used to build analyzers for nine languages. However, it has been criticized for being "too committed to a particular approach to linguistic analysis and representation" (Cunningham et al., 1997). ALEP's Text Handling component (Declerck, 1997) uses a particular SGML-based annotation that can be enriched with user-defined tags. Some standard components are provided, and rules allow the mapping of SGML tags to AVMs ("lifting"). SRI's *Open Agent Architecture (OAA)* [4] (Martin et al., 1999; Cheyer and Martin, 2001) is a software platform that offers a library for distributed agent implementation with bindings for several programming languages (C/C++, Java, LISP, PROLOG etc.). Agents request services from service agents via *facilitation*, a coordinating service procedure of transparent delegation, whereby facilitators can consider strategic knowledge provided by requesting agents, trying to distribute and optimize goal completion. Control is specified in a PROLOG-like Interagent Communication Language (ICL), which contains, but separates, declarative and procedural knowledge (*how to do* and *what to do*).

2.6 Discussion

Framework or Toolkit? The disadvantage of frameworks is that any such infrastructure is bound to have a steep learning curve (how to write wrapper/glue code, understand control) and developers are often reluctant to adopt existing frameworks. Using one frameworks also often excludes using another (due to the inherited "design dogma").

Toolkits, on the other hand, are typically smaller and easier to adopt than frameworks and allow for more freedom with respect to architectural choices, but of course the flip-side of this coin is that toolkits offer less guidance and reuse of architecture and infrastructure. See (Menzel, 2002) for a further discussion of architectural issues in NLP.

2.7 Design reuse with design patterns

Design patterns (Gamma et al., 1994; Harrison et al., 2000) are reusable units of software architecture design that have emerged from object-oriented software development research, where certain collaborative object configurations were found to re-occur in different contexts.

Finite-State Automata (FSAs) were historically the first devices that have received a software engineering treatment (Watson, 1995), as they are pervasive from compiler technology to software engineering itself. (Yacoub and Ammar, 2000) describe how using a `FiniteStateMachine` design pattern that separates out certain *facets* can facilitate interoperability between Mealy, Moore and hybrid FSAs.

(Manolescu, 2000) identifies the `FeatureExtraction` pattern as a useful abstraction for information retrieval and natural language processing: a `FeatureExtractorManager` is a `Factory` of `FeatureExtractor` objects, where each knows a `MappingStrategy`, a `FilteringStrategy` and a `Database`. Numerical techniques often used in machine learning to overcome the "curse of dimensionality" (→data sparseness above) such as *Singular Value Decomposition*, *Latent Semantic Indexing*, or *Principle Component Analysis* (PCA) are also instances of this pattern. It is worth noting that some of these patterns are domain-specific, i.e. the software engineering aspects interact with the type of linguistic processing. (Basili et al., 1999) generalize over typical NLP components, combining Data Flow Diagrams for a *Linguistic Processing Module (LM)*, a *Lexical Acquisition Module (LAM)* and an *Application Module (AM)* to a generic model of an NLP application. The result of the LAM is what (Cunningham et al., 1997) would call a *Data Resource* (as opposed to a *Processing Resource*, which corresponds to a LM). (Basili et al., 1999) also present an UML model of a class for linguistically annotated text, `LinguisticInformation`, that is interoperable with application-dependent classes.

2.8 Productivity gain with composition languages?

Recently, work in software engineering has focused on *composition languages* (Nierstrasz and Meijler, 1994), which allow to construct systems on a meta-level by specifying composition transformations in a separate glue notation without editing component source code (Aßmann, 2003). Such an approach would support a view held by (Daelemans et al., 1998), who argue that *"all NLP tasks can be seen as either*

[4] http://www.ai.sri.com/ oaa/

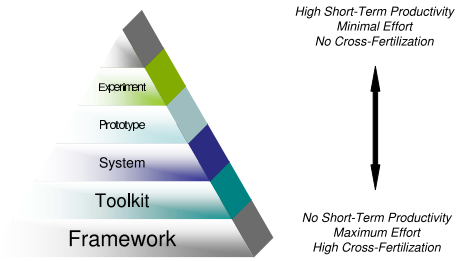

Figure 3: Productivity Pyramid.

- *light NLP tasks involving disambiguation or segmentation locally at one language level or between two closely-related language levels; or as*
- *compositions of light NLP tasks, when the task surpasses the complexity of single light NLP tasks.*"

That NLP processing often involves generic preprocessing (such as POS-tagging) can be taken as evidence for the need for dedicated linguistic composition languages.[5] Whereas toolkits and frameworks for NLP have already been developed, to date there exists no dedicated NLP composition language. In such a language, both linguistic structures (such as typed AVMs) and processing resources (such as taggers or tag-set mappers) had first-order status. Composition languages are a logical next step in the ongoing development of new abstraction layers for computing.[6]

3 Experiment or System?

Figure 3 depicts the trade-off researchers have to face when deciding between carrying out an experiment, building a prototype program, implementing a more fleshed-out self-contained system, building a complete, generic, redistributable toolkit or whether they invest long-term in providing the community with a new framework.[7] On the one hand, experiments ensure high short-term productivity with hardly any reuse or cross-fertilization to other projects. Frameworks, on the other hand, which are only possible in larger groups and with long-range funding, pay back relatively late, but offer many synergies due to their all-embracing nature if they can overcome developers reluctance to adopt a new framework.

[5] The visual application builder part of GATE 2 can be seen as a visual composition language.

[6] See (Abelson and Sussman, 1996) for a view that programming is indeed constant development and application of a growing collection of abstraction mechanisms.

[7] There may be a difference of several orders of magnitude in complexity between the tip and the bottom of the pyramid in Figure 3.

4 Summary and Conclusion

Whereas the evaluation of effectiveness of NLP methods has become an integral part of research papers, architectural evaluation is often neglected. It should also be recognized as vital part of engineering research publications, including an assessment of standard compliance, rapid deployability, maintainability and flexibility of design (Nyberg and Mitamura, 2002). Researchers should strive toward development of component APIs rather than prototypes to foster cross-fertilization and reuse. Frameworks are a valuable asset on the way as they embody common assumptions, but (unlike toolkits) they are not normally inter-operable with other frameworks. Already the horizon, NLP composition languages and could be an attractive solution to problems of productivity and reuse.

Acknowledgments. The author would like to acknowledge the financial support of the German Academic Exchange Service (DAAD) under grant D/02/01831 and of Linguit GmbH (research contract UK-2002/2).

References

Harold Abelson and Gerald Jay Sussman. 1996. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 2nd edition.

Uwe Aßmann. 2003. *Invasive Software Composition*. Springer, Berlin.

Roberto Basili, Massimo Di Nanni, and Maria Theresia Pazienza. 1999. Engineering of IE systems: an object-oriented approach. In Maria Theresia Pazienza, editor, *Information Extraction. Lecture Notes in Artificial Intelligence (LNAI 1714)*, pages 134–164. Springer, Berlin.

K. Bontcheva, H. Cunningham, V. Tablan, D. Maynard, and H. Saggion. 2002. Developing reusable and robust language processing components for information systems using GATE. In *Proceedings of the Third International Workshop on Natural Language and Information Systems (NLIS'2002)*.

Andrew Bredenkamp, Thierry Declerck, Frederik Fouvry, Bradley Music, and Axel Theofilidis. 1997. Linguistic engineering using ALEP. In *Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP'97)*, Sofia, Bulgaria, September.

Adam Cheyer and David Martin. 2001. The Open Agent Architecture. *Journal of Autonomous Agents and Multi-Agent Systems*, 4:143–148.

B. Crysmann, A. Frank, B. Kiefer, H. Krieger, S. Mller, G. Neumann, J. Piskorski, U. Schfer, M. Siegel, H. Uszkoreit, and F. Xu. 2002. An integrated architecture for shallow and deep

processing. In *Proceesdings of the Association for Computational Linguistics 40th Anniversary Meeting (ACL'02)*, University of Pennsylvania, Philadelphia, July.

H. Cunningham, Y. Wilks, and R. Gaizauskas. 1996. GATE – A General Architecture for Text Engineering. In *Proceedings of the 16th Conference on Computational Linguistics (COLING'96)*, Copenhagen.

H. Cunningham, K. Humphreys, R. Gaizauskas, and Y. Wilks. 1997. Software infrastructure for natural language processing. In *Proceedings of the Fifth Conference on Applied Natural Language Processing (ANLP'97)*, pages 237–244, Washington.

H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. 2002. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02)*, Philadelphia.

H. Cunningham. 2000. *Software Architecture for Language Engineering*. Ph.D. thesis, Department of Computer Science, University of Sheffield, Sheffield, UK.

Walter Daelemans, Antal van den Bosch, Jakub Zavrel, Jorn Veenstra, Sabine Buchholz, and Bertjan Busser. 1998. Rapid development of NLP modules with Memory-Based Learning. In *Proceedings of ELSNET in Wonderland*, pages 105–113.

Thierry Declerck, Alexander Werner Jachmann, and Hans Uszkoreit. 2000. The new edition of the natural language software registry (an initiative of ACL hosted at DFKI). In *Proceedings of the 2nd International Conference on Language Resources and Evaluation (LREC'2000), May 31–June 2*, Athens, Greece. ELRA.

Thierry Declerck. 1997. An interface between text structures and linguistic descriptions. In Ellen Christoffersen and Bradley Music, editors, *Proceedings of the Datalingvistisk Forening (DALF'97), June 9-10*, pages 8–22, Kolding, Denmark.

Bj¨orn Gamb¨ack and Fredrik Olsson. 2000. Experiences of language engineering algorithm reuse. In *Proceedings of the First International Conference on Language Resources and Evaluation (LREC'00)*.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

G¨unther G¨orz, Markus Kesseler, J¨org Spilker, and Hans Weber. 1996. Research on architectures for integrated speech/language systems in Verbmobil. In *Proceedings of the Conference on Computational Linguistics (COLING'96)*, Copenhagen.

Frank Griffel. 1998. *Componentware. Konzepte und Techniken eines Softwareparadigmas*. dpunkt, Heidelberg.

Ralph Grishman. 1995. TIPSTER Phase II architrecture design document (Tinman architecture). version 2.3. Technical report, New York University, New York.

Neil Harrison, Brian Foote, and Hans Rohnert, editors. 2000. *Pattern Languages of Program Design 4*. Addison-Wesley, Reading, MA.

Ken Lunde. 1999. *CJKV Information Processing*. O'Reilly, Sebastopol, CA.

Dragos-Anton Manolescu. 2000. Feature extraction: A pattern for information retrieval. In Harrison et al. (Harrison et al., 2000), pages 391–412.

D. Martin, A. Cheyer, and D. Moran. 1999. The Open Agent Architecture: a framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1/2):91–128.

Diana Maynard, Valentin Tablan, Hamish Cunningham, Cristian Ursu, Horacio Saggion, Kalina Bontcheva, and Yorick Wilks. forthcoming. Architectural elements of language engineering robustness. *Natural Language Engineering. Special Issue on Robust Methods in Analysis of Natural Language Data*.

Wolfgang Menzel. 2002. Architecture as a problem of information fusion. In *Proceedings of the International Symposium Natural Language Processing between Linguistic Inquiry and Systems Engineering*, pages 74–84, Hamburg.

Andrei Mikheev, Claire Grover, and Marc Moens. 1999. XML tools and architecture for named entity recognition. *Journal of Markup Languages: Theory and Practice*, 1:89–113.

Oscar Nierstrasz and Theo Dirk Meijler. 1994. Architecture as a problem of information fusion. In *Proceedings of the ECOOP'94 Workshop on Models and Languages for the Coordination of Parallelism and Distribution*.

Eric Nyberg and Teruko Mitamura. 2002. Evaluating QA system on multiple dimensions. In *Proceedings of LREC 2002 Workshop on QA Strategy and Resources*, Las Palmas, Gran Canaria, Spain.

Ben Shneiderman. 1997. *Designing the User Interface*. Addison-Wesley, Reading, MA, 3rd edition.

Neil Simpkins and Marius Groenendijk. 1994. The ALEP project. Technical report, Cray Systems / CEC, Luxembourg.

Ian Sommerville. 2001. *Software Engineering*. Addison-Wesley, Reading, MA, 6th edition.

Tuthill-Smallberg. 1997. *Creating Worldwide Software*. Sun Microsystems Press, Mountain View, CA.

Bruce W. Watson. 1995. *Taxonomies and Toolkits of Regular Language Algorithms*. Ph.D. thesis, Department for Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands.

Sherif M. Yacoub and Hany H. Ammar. 2000. Finite state machine patterns. In Harrison et al. (Harrison et al., 2000), pages 413–443.

Remi Zajac, Mark Casper, and Nigel Sharples. 1997. An open distributed architecture for reuse and integration of heterogeneous NLP components. In *Fifth Conference on Applied Natural Language Processing (ANLP'97)*.