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.

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 2.1 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 2.2 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). 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. Like any software, 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.

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

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

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

The 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 (http://www.ltg.ed.ac.uk/software/xml/) 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.

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 Verb-mobil 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) (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 (Altmann, 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

http://www.ai.sri.com/oaa
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.

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 \cite{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


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. 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.

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.\(^5\) On the one hand, experiments ensure high short-term productivity with hardly any reuse or cross-fertilization to other projects. Frameworks, on the other

\(^5\) The visual application builder part of GATE 2 can be seen as a visual composition language.

\(^6\) See \cite{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.


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


