Component-based Development of Software Language Engineering Tools

JACKLINE SSANYU AND KEES HEMERICK[*]
Jackline Ssanyu, Department of Computer Science, Kyambogo University.
Kees Hemerick, Software Engineering & Technology Group, Department of Mathematics and Computer Science. Technische Universiteit Eindhoven.

ABSTRACT

In this paper we outline how Software Language Engineering (SLE) could benefit from Component-based Software Development (CBSD) techniques and present an architecture aimed at developing a coherent set of lightweight SLE components, fitting into a general-purpose component framework. In order to give an impression on our development style, in this paper we demonstrate how to compose a syntax highlighter from a set of available SLE components using the NetBeans environment. Developing SLE tools as lightweight components that fit into general-purpose frameworks has advantages over the usual trend in which SLE tool development is towards large special-purpose frameworks. It facilitates incorporation of language processing tasks into all kinds of applications and makes SLE techniques available to occasional or first-time users with little effort.

Categories and Subject Descriptors D.2.11 [Software Engineering]: Software Architectures—Data abstraction; D.2.3 [Software Engineering]: Reusable Software; D.2.6 [Software Engineering]: Programming Environments—Graphical environments; D.3.2 [Programming Languages]: Language Classifications—Specialized application languages;
D.3.4 [Programming Languages]: Processors—Translator writing systems and compiler generators

Additional Keywords Component-based Software Development, Language Engineering, Component Frameworks, Composition, SLE Architecture

IJCIR Reference Format:

Ssanyu ,Jackline and Hemerick, Kees. Component-based Development of Software Language Engineering Tools. International Journal of Computing and ICT Research, Special Issue Vol. 5, Special Issue, pp 7-16. http://ijcir.org/specialissue2011/article2.pdf

1. INTRODUCTION AND MOTIVATION

Software Language Engineering (SLE) is a recently coined name [SLE] for an old and well-established field, viz. that of methods and tools for manipulation of artificial languages in software engineering. The field has an

[*] Jackline Ssanyu, Department of Computer Science, Kyambogo University. P.O. Box 1, Kampala, Uganda. Email: j.ssanyu@tue.nl
Kees Hemerick, Software Engineering & Technology Group, Department of Mathematics and Computer Science. Technische Universiteit Eindhoven. P.O. Box 513, 5600 MB Eindhoven, The Netherlands. Email: c.hemerik@tue.nl

International Journal of Computing and ICT Research, ISSN 1818-1139 (Print), ISSN 1996-1065 (Online), Vol.5, Special Issue, pp 7-16. December, 2011.

extensive body of solid theory on subjects like grammars and automata [Hopcroft et al. 2006], a wealth of algorithms, a large collection of tools, and a broad range of applications. Tools range from the classic Lex [Lesk and Schmidt 1975] and Yacc [Johnson 1975] and their follow-ups to sophisticated meta-environments aimed at generation of entire compiler front-ends or Integrated Development Environments (IDEs) such as Stratego/XT, Eclipse IMP, ELI, The Meta-Environment and ANTLR [Stratego; Eclipse; Eli; Meta-Env; ANTLR].

Component-Based Software Development (CBSD) is a major approach for software development. CBSD aims to build software systems by combining and configuring existing software components. Such an approach helps to build systems more rapidly and reduces software development costs by composing a system from existing components instead of building it from scratch. It also enables software reuse since components can be reused in developing many systems. CBSD is based on the assumption that certain parts of software systems reappear in many applications and that software should be assembled through reuse rather than rewriting. Each software component is dedicated to a particular task (i.e. can work as an independent unit) and has (a) well defined interface(s), appropriate documentation and a defined reuse status. Software components fit into a so-called component framework, an architecture which provides facilities for customization of and cooperation between components [Szyperski 2002]. Some well-known component frameworks are NetBeans [NetBeans], Eclipse [Eclipse], Microsoft .NET framework [Microsoft] and Delphi with its Visual Component Library (VCL) [Delphi].

In principle, the SLE field lends itself well to application of component-based methods because:

- It consists of well-defined tasks such as scanning, parsing, tree building, syntax highlighting and others [Aho et al. 2006]. We can define the tasks as components (with simple interfaces), and embed them into a general-purpose component framework to be combined in different ways while building applications for different purposes.
- Each task has good theory and a variety of well-studied algorithms for implementing it [Aho et al. 2006; Aho and Ullman 1972; Grune and Jacobs 1990; Wilhelm and Maurer 1995]. We can define the existing algorithms as components so that users have a variety of components per task at their disposal.

Historically speaking, much work on SLE theory and tooling has preceded work on CBSD, and therefore existing SLE tools have not utilized the full benefits [Szyperski 2002] of CBSD. Many SLE tools are incorporated in special-purpose systems [Stratego; Meta-Env; ANTLR]. Although these systems are very effective in the hands of experts, they can be rather complex and inaccessible to non-experts or occasional users that have modest knowledge of SLE technology.

We are investigating how development of SLE tools can benefit from consciously exploiting CBSD techniques. Rather than designing a large special-purpose framework, our main goal is to develop a coherent set of small SLE components and generators, where each component is dedicated to a single well-defined SLE task. The components should fit into a general-purpose component framework like NetBeans, and it should be possible to manipulate them inside an IDE like NetBeans or Eclipse, just like any other component.

We do so for the following reasons:

- **Lightweight** - Many kinds of applications do some language processing as a necessary but subordinate task. For instance, an interactive theorem prover does some scanning, parsing, tree building and formatting of logical formulae, but its core business is proof construction. In such cases, using a large special-purpose SLE framework for the SLE tasks is overkill and distracts from the main tasks. Using some lightweight SLE components seems a better solution.
- **Ease of use** - Many software developers use an IDE which supports a drag-and-drop style for application development from components. This style is very effective for design of graphical user interfaces, but can also be used for handling non-visual or
  domain-specific components. In fact, the IDE can handle any components that comply with the supported component model in a uniform way. Thus, when a developer has a set of compliant SLE components available, he can use SLE technology without the problems of learning to use a large special-purpose SLE environment and relating it to his application.
- **Variety** - Having at one's disposal a variety of components for a certain SLE task (e.g. several parser components with the same interface but parsing according to different strategies) makes it easier to experiment with different techniques and to pick the component most suited for the task at hand. This can be a great asset in an educational environment.

To achieve our goal, the following research problems have to be solved:

1. Design the architecture for the SLE components and their interactions.
2. Study and selection of algorithms to be implemented as SLE components.
3. Describing the algorithms in a uniform way, making use of common data structures etc.
4. Realization of the SLE components in a concrete development environment like NetBeans to demonstrate the usability of our design.

In this paper we focus on the first and fourth research questions.

The paper is organized as follows: In Section 2, we present the architecture of our SLE components. In section 3 we illustrate the use of our SLE components in the form of a step-by-step development of a syntax highlighter. In Section 4, we explain how our SLE component framework differs from the present SLE tools. In Section 5, we describe current status and future work.

2. SLE COMPONENTS ARCHITECTURE

The SLE component framework supports software development styles (i.e. drag-and-drop and wizards) commonly found in modern IDEs like NetBeans, Eclipse and Delphi. It is also geared towards the following front-end SLE tasks:

- **Lexical Scanning** - the process of partitioning an input string into consecutive substrings such that each substring represents a token according to some lexical grammar. Typically, tokens are reserved words, identifiers, numbers, punctuation characters, single and multi-character operators, comments etc. The main approaches to lexical scanning are based-on: Finite Automata [Aho et al. 2006; Thompson 1968] and ELL (1) recognition programs [Lewi et al. 1979; Wirth 1976].
- **Parsing** - the process of analyzing an input string in order to determine its grammatical structure (syntax tree) with respect to a given context free grammar. Traditionally, there are two categories of parsing techniques: Top-down (LL) and Bottom-up (LR) parsing. Good descriptions of the techniques can be found in [Aho et al. 2006; Grune and Jacobs 1990].
- **Tree building –** process of constructing a tree (e.g. concrete syntax tree (parse tree) and/or abstract syntax trees (ASTs)) from a set of nodes. Tree building may be top-down or bottom-up.
- **Flattening –** the translation from a syntax tree to a sequence of tokens.
- **Pretty printing –** transformation of a syntax tree to a well formatted string.

Generally, the SLE framework provides a collection of components that deal with transformation of language terms from a concrete textual form to abstract syntax trees (ASTs) and the converse transformation from abstract to concrete. This includes components for:

- Representing language terms in various forms (plain text, sequence of attributed symbols, parse tree, abstract syntax tree, etc.);
- Standard transformations between these forms (lexical scanning, parsing, tree building, pretty printing and flattening)
- Viewing and editing the various term representations and other related data (scan tables, coloring schemes, etc.)

Somewhat unusual, it also contains components for representing language specifications (lexical syntax, context-free syntax, abstract syntax, etc.). Other components can observe such a language specification component and adapt their own state and behavior to it. This mechanism is known as the Observer design pattern [Gamma et al. 1995] and will be described in more detail in the rest of this section.

In the following subsections 2.1 - 2.5 we shall describe the intended use of our SLE components, their general characteristics, component interfaces, data flow management between SLE components and composition. In subsection 2.6 we discuss compatibility of the SLE component model with some existing component models.

2.1 Intended Use

The SLE component framework supports development of language front-ends in ways commonly found in modern IDEs like NetBeans, Eclipse and Delphi:

- It allows a drag-and-drop development style for quick and easy construction of large parts of a language front-end with little or no coding. Components are dragged from a palette and dropped on a form; their properties are set by means of property editors and/or customizers.
- It provides wizards that allow a user to rapidly generate language-specific pieces of code. The user is guided through a multi-step dialog to set his preferences, which are then used to customize some existing SLE components and/or generate some specialized source code-based SLE components.

2.2 General Characteristics of SLE Components

The general characteristics common to all SLE components are:

- **Properties** - All SLE components have properties through which their appearance and behavior can be customized. Properties range from standard types (such as string, integer, font) which can be edited using property editors, to SLE-specific data, such as grammars or formatting rules. We provide customized editors for editing such data at building (design) time. However, data can also be edited at run time via *getter* and *setter* methods. Properties may also be references to other SLE components.
- **Persistency** - SLE components are persistent. Their state can be customized in the IDE at design time and then saved to storage and reloaded later.

- **Composition** - SLE components may have recursive structure, following the Composite design pattern [Gamma et al. 1995]. In this way, a group of cooperating SLE components may be turned into a new SLE component.
- **Event handling** - An SLE component can be both observer and observable of other SLE components, in the sense of the Observer design pattern [Gamma et al. 1995].

SLE components may observe a state change in another SLE component and adjust their own state accordingly.

- **(Non-) visual** - There are both visual and non-visual SLE components. Typically, non-visual components hold data like text, parse tables or parse trees, whereas visual components provide certain views and editing capabilities, such as (language specific) text editors, tables or tree views.

2.3 Component Interfaces

SLE Components have simple well-defined interfaces. Each SLE task has an interface to capture the essentials of that task and one or more components implementing that interface. For instance, for the parsing task, the interface would provide facilities for recognizing the main syntactic categories and for communicating with a scanner and a tree builder. For each of these interfaces, there may be several components implementing that interface according to different strategies. For instance, for scanning there could be both a Lex-like scanner based on finite automata [Aho et al. 2006; Thompson 1968] and an ELL (1)-based scanner [Lewi et al. 1979; Wirth 1976].. For parsing there could be parsers using simple and efficient strategies (such as recursive descent (LL) and Simple LR (SLR) [Wilhelm and Maurer 1995] as well as parsers employing more general methods like Generalized LR (GLR) [Scott et al. 2000]. The visual environment aids in the process of linking the different SLE components together through their interfaces.

2.4 Data Flow between SLE Components

Our SLE framework implements the Observer design pattern [Gamma et al. 1995] to maintain consistency and to aid the flow of information between varieties of cooperating SLE components. Typically, when the state of an SLE component changes, its observers are informed about the state change so that they can adapt to it. The amount of information may vary widely. For small changes, a "push" model is used, whereby the observable component sends observers detailed information (e.g. symbol value to "while", color attribute for keywords changed to red) about the change, whether they want it or not. Then, the observers may adjust their own state directly. For more involved changes, a "pull" model is used in which the observable sends nothing but the most minimal notification, and observers ask for details explicitly thereafter ("pull" model).

This kind of data flow allows users (e.g. occasional and designers of small languages) to design and develop applications easily by adjusting properties of one component and all other SLE observer components are adjusted automatically. We explain one such interesting aspect: Consider a language specification (e.g. consisting of a lexical and a context-free grammar) held in a component and scanner and parser components with their own "hidden" generators which are observers of the language specification. When the language specification changes, scanners and parsers can adapt to the change immediately by calling a hidden generator implicitly.

As an example, consider the following configuration of SLE components, which together achieve syntax highlighting:

- **Language** - an observable component holding a lexical and context-free grammar;
- **Scanner** - a Lex-like table-driven scanner; it observes the *Language* component;
- **ColorScheme** - a mapping from the symbols of the language to font and color attributes; it observes the *Language* component;
- **RichTextView** - a text editor with text coloring capabilities; it uses both the *Scanner* and *ColorScheme* components and observes their changes.

When the lexical part of the *Language* component changes, a property change is observed by both the Scanner and the *ColorScheme* components. The following two scenarios take place:

- The *ColorScheme* component reacts to the observed property change of the *Language* component by adjusting its domain of valid symbols to that of the new language. The *ColorScheme* in turn sends a property change notification to its observers. In this case *RichTextView* observes the change and reacts by using the *Scanner* to scan its text and display the symbols recognized according to the new mapping in the *ColorScheme*.
- The *Scanner* component reacts to the change in the *Language* component by invoking its hidden scanner generator to regenerate its scan tables for the new language. The change in the *Scanner* is in turn observed by the *RichTextView*, which uses the adjusted scanner to highlight its text according to the new language.

In section 3.1 we return to this example and elaborate it in the concrete setting of the NetBeans IDE.

2.5 Composition of SLE Components

Components are meant for composition. Therefore, our SLE components have a recursive structure. In this way, a group of cooperating SLE components may be turned into a new SLE component, which may subsequently be used just like any other SLE component.

As an example, reconsider the syntax highlighting example of subsection 2.4. As this task may occur more often, a new component *SyntaxHighlighter* might be constructed for it, containing the *Scanner*, *ColorScheme* and *RichTextView* as subcomponents. The new *SyntaxHighlighter* as a whole may be an observer of a Language component, whereas it internalizes the collaboration between *Scanner*, *ColorScheme* and *RichTextView*.
In section 3.2 we return to this example and elaborate it in the concrete setting of the NetBeans IDE.

2.6 Compatibility with Existing Component Models

The SLE component model above has been described without reference to existing programming languages, component models, or IDEs. It can be realized in various environments. In the following subsections, we discuss two such realizations.

2.6.1 Java, JavaBeans and NetBeans

The JavaBeans APIs define a software component model for the Java programming language. According to the JavaBeans API specification [Java Beans 1997]:

"A JavaBean is a reusable software component that can be manipulated visually in a builder tool."

The typical distinguishing features are:

- Support for *introspection* so that a builder tool can analyze how a bean works.
- Support for *customization* so that when using an application builder a user can customize the appearance and behavior of a bean.
- Support for *events* as a simple communication metaphor that can be used to connect up beans.
- Support for *properties*, both for customization and for programmatic use.
- Support for *persistence*, so that a bean can be customized in an application builder and then have its customized state saved away and reloaded later.

To create and use JavaBean components one can directly and easily make use of all these features via JavaBeans-compliant application builder tools such as Eclipse, JBuilder, NetBeans and others. In our research we focus on NetBeans and its GUI design tool. NetBeans allows users to rapidly construct applications by dragging and dropping components on a form. Users can combine components into applets, applications, or composite components. These components range from the default components from AWT and Swing libraries to user-developed components. All components are stored on the component palette. To construct an application, the user drags and drops one component at
a time from the palette on to the form, edits the desired component properties and makes appropriate connections among the components. Additionally, NetBeans provides a number of wizards that assist a user to rapidly build complex pieces of code.

Considering the above features of JavaBeans and their builder tools, JavaBeans provides an attractive component model for realizing most of the requirements of our SLE components. In particular, the most interesting aspects are:

- The JavaBean's events model [JavaBeans 1997] nicely accommodates the implementation of observer/observable behavior of our SLE components.
- Persistence of data elements is offered through automatic Java serialization mechanism [JavaBeans 1997].
- The Java interfaces give a nice implementation concept to realize the idea of interface definitions of our SLE components.

2.6.2 Object Pascal, VCL and Delphi

Delphi is a RAD (Rapid Application Development) environment based on the programming language Object Pascal. Among its features are: a visual Form Designer and a large Visual Component Library (VCL). The latter is essentially an Object Pascal layer on top of the Windows Win32 API. VCL components can be picked from a palette and placed on a form, have their properties edited using the Object Inspector, or can be edited as a whole by a Component Editor. VCL components have support for event handling and for persistence.

As this summary suggests, most of the facilities required by our SLE components can be realized in Delphi using the VCL and the Form Designer. The main points of attention are:

- Delphi's persistence mechanism - based on streaming published properties of VCL components to and from form files - is geared towards visual components and simple data types for properties. Streaming

non-published properties and properties of more complex types can be done, but involves some extra work.

- The Observer/Observable mechanism is not *a priori* available, but can easily be realized by having suitable classes in the SLE class hierarchy, implement the *IObserver* and *IObservable* interfaces.

3. CONCRETE EXAMPLE

In order to give an impression of the development style of our SLE component set and their intended usage, let us reconsider the syntax highlighting example in subsection 2.4 and illustrate how to assemble a simple syntax highlighter using the NetBeans IDE environment. When we open the NetBeans IDE (see Figure 1), we see on the component palette among others the following components that suit our purpose:

- **CLexicalGrammar** - This component holds a lexical grammar and maintains its consistency. It notifies its observers of changes in the grammar.
- **CDFAScanner** - This component is a lexical scanner, based on a DFA (Deterministic Finite Automaton). The component has a hidden scanner generator, based on the Lex algorithms [Lesk and Schmidt 1975]. The component also has a property *lexicalGrammar*, which is a reference to a *CLexicalGrammar* component. When the *CDFAScanner* component observes a change in the CLexicalGrammar component, it invokes its hidden scanner generator to regenerate its DFA tables and notifies its observers.
- **CColorScheme** - This component maps grammar symbols to symbol categories and symbol categories to font and color attributes. It observes a *CLexicalGrammar* component and maintains consistency between its own valid symbol domain and the symbols defined in the grammar.
- **CRichTextView** - This component is a text editor with facilities for different font and color attributes. It uses a *CDFAScanner* component to scan its text and a *CColorScheme* component to render the recognized symbols. When it observes changes in scanner or color scheme it re-renders its text.

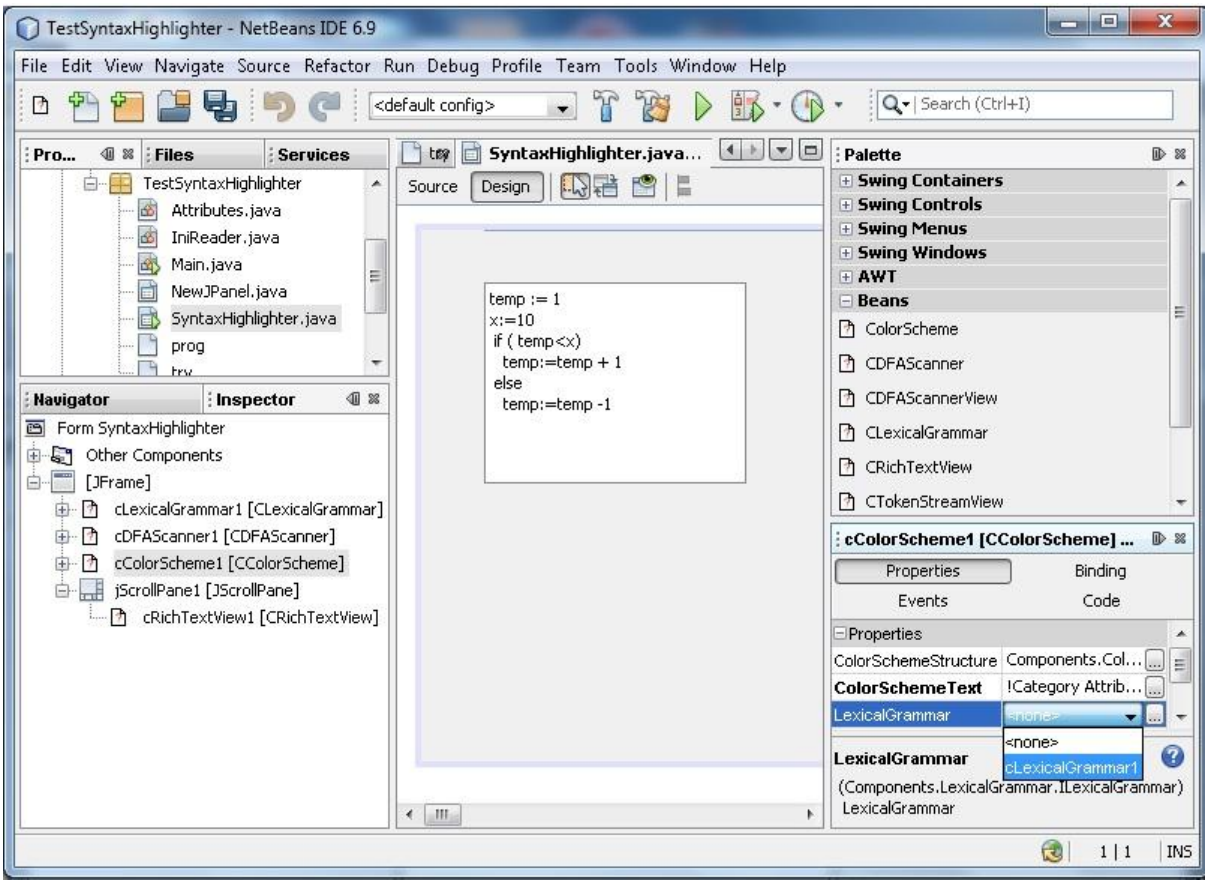

Figure 1: The NetBeans IDE in design mode shows the component palette in the right upper window. The middle window contains a form on which SLE components can be dropped. Visual components may be accessed on the form itself; all components can be accessed from the Inspector window on the left lower side of the IDE. The right-lower window is a sample property editor for the Colorscheme component. The ColorScheme's *LexicalGrammar* property shows a value *cLexicalGrammar1* indicating that it is connected to the *CLexicalGrammar* component on the form.

3.1 Constructing a Syntax Highlighter from Existing Components

To construct a syntax highlighter from the SLE components mentioned we follow the following steps:

1. Select a *CLexicalGrammar* component from the Palette and drop it on the form. Note that *CLexicalGrammar* is a non-visual component, therefore it doesn't show on the form but can be accessed from the inspector window (See Figure 1) of the NetBeans IDE. Click the *CLexicalGrammar* component (from the inspector window) and use its property editor to set the grammar. The property editor shows two properties: *lexicalGrammarStructure* (represents the internal structure of Lexical definitions) is used to open a customizer for editing parts of the lexical grammar and *lexicalGrammarText* (represents the grammar in text form) is used to set the grammar in text form. Use any of the two properties to set the grammar. The following updates take place:
    - It checks that the grammar is well-formed and computes analysis information (i.e. *first, follow* and *nullable* sets).
    - Maintains consistency between l*exicalGrammarStructure* and *lexicalGrammarText* properties of the lexical grammar.
    - Sends a property change to any of its observers.

2. Select a *CDFAScanner* component from the Palette and drop it on the form. The component shows in the inspector window of the IDE. Connect it to the CLexicalGrammar component by setting its *lexicalGrammar* property to the instance of *CLexicalGrammar* placed on the form in step 1. If the connection is successful, the following actions take place:
    - *CDFAScanne*r is added to the list of observers of the *CLexicalGrammar* component.
    - *CDFAScanner* invokes its hidden scanner generator to construct scan tables that correspond to the lexical definitions of *CLexicalGrammar*.
    - Maintains consistency between the *CDFAScanner* properties, *DFATableText* (for textual representation) and *DFATableStructure* (internal representation).
    - A property change is sent to the components observing *CDFAScanner*, if any.

3. Select a *CColorScheme* component and drop it on the form. Access the component via the inspector window and open its property editor. Connect it to the *CLexicalGrammar* component by setting its *lexicalGrammar* property to the instance of *CLexicalGrammar* placed on the form in step1. Similarly, a successful connection *causes* the following actions:
    - The *CColorScheme* is added to the list of observers of the *CLexicalGrammar* component.
    - The component adjusts its domain of valid symbols to that of the grammar.
    - Maintains consistency between the *CColorScheme* properties *ColorSchemeText* (for textual representation) and *ColorSchemeStructure* (internal representation).
    - A property change is sent to the components observing *CColorScheme*, if any.

    Use the *ColorSchemeStructure* property to open a customizer (see figure 2) and edit the list of symbol categories, set their desired color and font attributes and classify symbols into their appropriate categories. Alternatively, use the *ColorSchemeText* property to edit the *CColorScheme* settings in text form. Either way, the *ColorSchemeText* or *ColorSchemeStructure* properties will be updated automatically and in turn a property change is sent to the components observing *CColorScheme*, if any.

4. Finally, select a *CRichTextView* component and drop it on the form. Enter text in its text property. Connect it to the *CDFAScanner* and *CColorScheme* by setting its scanner and *colorScheme* properties (to the components placed on the form in step 2 and step 3) respectively. *CRichTextView* component reacts in the following way:
    - It starts to observe changes in the *CDFAScanner* and *CColorScheme* components.
    - It uses both the *CDFAScanner* and *CColorScheme* components to highlight the text according to the lexical grammar defined in *CLexicalGrammar*.

Note that the syntax highlighter has been constructed without writing a single line of code.

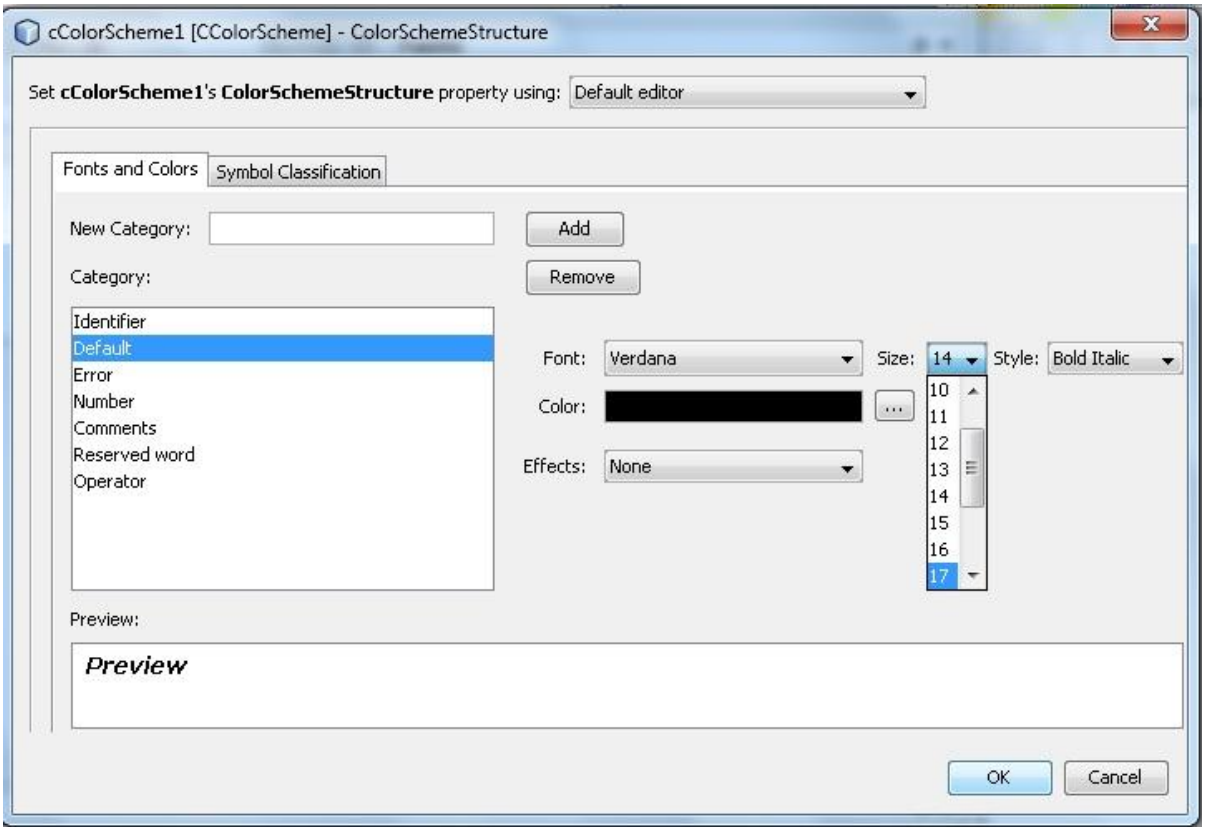

Figure 2: An example of a customizer for the ColorScheme component. Tab, Fonts and Colors lists the current symbol categories with their font and color properties. Tab, Symbol Classification lists the symbols supported by the syntax highlighter and their corresponding categories.

3.2 Making the Syntax Highlighter a New Component

With only slightly more effort, a syntax highlighter like the one developed above can be turned into a new component. We sketch the steps:

1. This time, start with a JPanel (a visual component that can contain other components);
2. Drop on it a *CDFAScanner*, a *CColorScheme* and a *CRichTextView*, and connect them as above;
3. Switch to the code view, and add two public properties to the JPanel: *language*, which is a reference to a *CLexicalGrammar* component, and *text* , which is a String. A few lines of code have to be added for the setters and getters of these properties. The setter of the *language* property has to pass its value on to the *language* property of both the *CDFAScanner* and the *CColorScheme* sub-components. The setter and getter of the *text* property will pass a string to and from the *text* property of the *CRichTextView* respectively.
4. Register the newly created component on the Palette. Thereafter it can be used just like any other SLE component.

4. RELATED WORK

The idea of composing a language processing system from subsystems dedicated to specific tasks like scanning and parsing is an old one [Bauer and Eickel 1974]. A lot of research
has gone in generating such subsystems from language specifications. Some of these tools, like Lex and Yacc [Lesk and Schmidt 1975; Johnson 1975], were designed to cooperate with each other and with other subsystems. They could be considered components *avant la lettre*.

Stratego/XT [Stratego] is a collection of components for transformation tasks, like parsers and pretty-printers. The XT components are executable tools: they can be executed from the command line and they can be composed into a pipeline. The tools in the pipeline exchange structured data in the form of annotated terms. The tools themselves can be implemented in different programming languages.

There exist several systems that aim to make language tools easier to use and easier to integrate in common programming environments. A well-known example is Eclipse IMP (IDE Meta-Tooling Platform) a project to support the development of richly-featured, language-specific IDEs in Eclipse [Eclipse]. It provides a.o. a set of language service-creation wizards, a run-time framework to encapsulate common language-processing infrastructure, and DSLs for easily implementing certain IDE services.

The components we propose are not separate executables and do not come with an elaborate infrastructure. They conform to standard component models like JavaBeans or the Delphi VCL and can be handled by any environment supporting such a component model. To the best of our knowledge, such a component collection does not exist yet.

5. CONCLUSION AND FUTURE WORK

The research problems mentioned in section 1 are the subject of the PhD. Project of the first author. Starting point was an unfinished prototype in Delphi, developed by the second author.

As most of the design of an SLE component set can be done independently of a concrete platform, it was decided to make an abstract platform-independent design, accompanied by two concrete realizations: one for Java and NetBeans - developed by the first author and one for Object Pascal - developed by the second author. There is method to our madness: most problems concerning the design of a coherent set of algorithms and data-structures for SLE tasks are much clearer and simpler at an abstract level (compare for example the algorithms in a book like [Wilhelm and Maurer 1995] with the code of some existing parser generators. Based on an abstract design, implementation in Java and/or Object Pascal is relatively easy.

This way of working also helps in separating platform-independent aspects from platform-specific ones.

The SLE component architecture has been realized as a prototype and tested on various examples in both the NetBeans 6.9 and Delphi 2010 environments. The experiments do not only concern the drag-and-drop style illustrated in this paper, but also the use of wizards to generate source code for components as described in section 2.1, but not elaborated in this paper.

Until now, the focus of our work has mainly been on:

- Identifying suitable characteristics of the SLE components framework;
- Investigating the best way of maintaining consistency and managing information flow among a collection of SLE components;
- Investigating how to handle source code generating SLE components in both Netbeans and Delphi environments.

The SLE framework currently has a limited number of components and wizards. These include components and wizards for:

- Lexical grammars (including views, editing, and grammar analysis);
- Various lexical scanner components (including views and hidden scanner generators for both ELL(1) and DFA methods);
- Syntax highlighting.

Currently we are working on a similar collection for parsing, including a family of recursive descent parsers with and without backtracking. This will be followed by components for tree building and flattening.

REFERENCES

AHO, A. V, LAM, S.M, SETHI, R, AND ULLMAN, J. D. 2006. *Compilers: principles, techniques, and tools.* Addison-Wesley, Boston, ISBN: 0-321-48681-1

AHO, A. V, AND ULLMAN, J.D. 1972. *The Theory of Parsing, translation, and compiling.*, volume 1: Parsing. Prentice-Hall, Inc., ISBN: 0-139-14556-7

ANTLR. ANother Tool for Language Recognition, *Internet WWW page, at URL:* http://www.antlr.org/.

BAUER, F.L, and EICKEL, J. 1974. Compiler Construction - An Advanced Course. Springer LNCS 21.

DELPHI. Delphi. *Internet WWW page, at URL:* http://www.embarcadero.com/products/delphi.

ECLIPSE. Eclipse IMP: Ide Meta-tooling Platform, *Internet WWW page, at URL:* http://www.eclipse.org/imp/.

ELI. Eli project. *Internet WWW page, at URL:* http://eli-project.sourceforge.net/.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, ISBN: 0-201-63361-2

GRUNE, D, AND JACOBS, C. 1990. *Parsing Techniques-A Practical Guide*. Monographs in Computer Science. Ellis Horwood Limited, Chichester, England.

HOPCROFT, J. E, MOTWANI, R, AND ULLMAN, J. D. 2006. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, ISBN 0-321-46225-4

JAVABEANS. 1997. JavaBeans API Specification, *Internet WWW page, at URL: http://java.sun.com/beans.*

JOHNSON, S.C. 1975. Yacc: Yet another compiler-compiler. Comp. Sci. Tech. Rep. Bell Laboratories, Murray Hill, NJ.

LESK, M. E., AND SCHMIDT, E. 1975. Lex - a lexical analyzer generator. Comp. Sci. Tech. Rep. 39, Bell Laboratories, Murray Hill, NJ.

LEWI, J, DE VLAMINCK, K, HUENS J, and HUYBRECHTS,M. 1979. *A Programming Methodology in Compiler Construction*, Part 1 and 2, North-Holland Publishing, Amsterdam.

META-ENV. The Meta-Environment, *Internet WWW page, at URL:* http://www.meta-environment.org/.

MICROSOFT. The .NET Framework, *Internet WWW page, at URL:* http://www.microsoft.com/net/.
NETBEANS. NetBeans IDE, *Internet WWW page, at URL:* http://netbeans.org/.
SCOTT, E, JOHNSTONE, A, and HUSSAIN, S.S. 2000. Tomita-Style Generalised
LR Parsers. TR-00-12, University of London, Department of Computer Science, Royal Holloway, London, *Internet WWW page, at URL:h*ttp://www.cs.rhul.ac.uk/research/languages/publications/tomita_style_1.ps.
SLE. International Conference on Software Language Engineering. *Internet WWW page, at URL: http://planet-sl.org/.*
STRATEGO. Stratego/XT, *Internet WWW page, at URL:* http://strategoxt.org/.
SZYPERSKI, C. 2002. *Component Software- Beyond object-oriented programming*. Addison Wesley, ISBN 0-201-74572-0
THOMPSON, K. 1968. Programming techniques: regular expression search algorithm. Communications of the ACM, New York, NY, USA, vol.11(no.6), 419-422.
WILHELM, R, AND MAURER, D. 1995. *Compiler Design*. International Computer Science Series. Addison-Wesley, ISBN: 0-201-42290-5
WIRTH N. 1976. *Algorithms + Data Structures = Programs.* Prentice Hall, NJ, USA, ISBN: 0-130-22418-9