

# Keyword based programming: Languages at your feet

T. Cleenwerck & E. Duval, H.Olivie  
K.U.Leuven  
Celestijnenlaan 200A, Belgium  
thomas@cs.kuleuven.ac.be

Keywords: domain specific languages, reuse

Classification: one year's work.

## 1. Introduction

Programming is building up knowledge that should be documented, reused, evaluated and improved. Why documented? Because code written in today's programming languages is not a good communication channel. Why reused? Because of the enormous advantage of not inventing things twice. Why evaluated? Because that's the moment where we can improve our-selves. There are not many people out there who practice this schema on a daily basis. Most of us start off on the wrong foot. Writing good documentation takes time and once it is done, it must be updated whenever your solution changes. Documentation is there to state our intention that is scattered over a piece of code. Knowledge in the form of intentions and abstractions made in code is thus hard to find. Using extensible languages the programmer can write clear, intentional and self-documenting code.

It is only a slight exaggeration to say that every good comment in a program represents a small failure of the language [Sim95].

When extensible languages are used extensively, and they should, domain specific elements will emerge in them and languages will adapt to their environment becoming domain specific languages (DSLs). DSLs are separating the problem and solution space; allowing solutions to be discussed, exchanged, evaluated and improved independently. In this perspective the "word" domain is not only interpreted as an application field [SCK+96], but the "word" domain can be put in a much broader context covering every environment where there is some kind of programming activity, capturing a single programmer *learning* how to translate requirements into running code. This notion of domain has been traditionally used in OO, AI and knowledge engineering communities. The UML (Unified Modelling Language) glossary gives the following definition [BRJ99].

Domain: An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.

Only once in a while domain specific languages really escape the realm of creation and are used widely (HTML, LaTeX, XSLT, JSP, Risl). We will use the following terminology:

*Domain Specific Language (DSL)* A small, usually declarative, language expressive over the distinguishing characteristics of a set of programs in a particular domain [WAL96].

*Domain Specific Description (DSD)* A "program" (specification, description, query, process, task, ...) written in a DSL.

*Domain Specific Processor (DSP)* A software tool for compiling, interpreting, or analysing domain specific descriptions.

Few DSLs are built with a proper environment and tool support. In fact, some DSL's are still built from scratch. It is thus not surprising that a lot of programming experience is required in the domain, and a risk analysis is performed before even considering developing and using a DSL. Building DSLs requires nowadays several skills like mastering compiler technology, knowing which sort of problems and their solutions are frequently occurring in the domain and developing a language in which domain specific solutions can be expressed. In this paper we present a development environment where domain specific languages of any kind can be built in an incremental and experimental way. The aim is to make the development of DSLs significantly easier by reducing the implementation effort for lexers and parsers and by providing a single integrated environment where the different development tools can be used transparently and declaratively.

The DSD in which programs are to be written contains a highly declarative description of the problem allowing enough variability, for the aim is to build several applications of the same family. It is a specification for a solution to be generated automatically. The solutions are expressed in a DSP. In other words, DSPs contain *active knowledge* of how to interpret the specification.

In our development environment *keywords* are used as the basic building blocks of a DSL. Keywords are active abstractions:

- Containing a semantic meaning, describing what the abstraction stands for, its function as a language component, when and how it can be used, etc in a DSL.
- Containing a syntactical description, describing what to write, to use or to activate the abstraction in a DSD.
- Active because they contain code to execute the abstraction and to generate solutions from the abstract specification in a DSP.

A DSL can be perceived as a *composition* of such keywords in which you are allowed to think and program in domain specific constructs.

## 2. Keyword Based Programming

As mentioned in the introduction, constructing DSLs is not easily done. In this section we will discuss the environment we've developed in order to support semi-automated compiler construction. Our environment allows incremental and experimental building of proprietary languages, enabling rapid and accurate (because of the early feedback) development. Our environment consists of a couple of basic building blocks integrated into a small application. More specific layers can be built around it. It is our intention to provide a common environment where languages of any kind can be built, ranging from aspects, subjects, intentions, XML and SQL processors to scripting languages.

A DSL is composed of two parts:

- *Keywords*. Active language components, which are described in another proper DSL to minimize the programming effort.
- *Language specification*. Keywords are composed into a language. The specification is the glueware to customize the basic building blocks and integrate them into one single language.

The benefits of this decomposition lay's in the reusability of the basic language components rendering your DSL into a *plug & play domain specific language*.

### 2.1. Language specification

A DSL has a name and a path. The generated compiler will be stored in the given path. The package is an implementation detail, because the generated compiler is a java program. The language has a number of *keyword trees*. The first keyword tree is called the *root* (e.g. the class keyword in java) and is the starting point of the DSL. A *keyword* is an active component usually one abstraction that is responsible for executing the its own semantic effect. Examples of keywords are an if-then-else-construction, a method declaration and a timer. The time keyword will time how long it takes to execute a piece of code. The code below illustrates the usage of the keyword: in the method `stackTest` the keyword `time` will track how long it takes to push a hundred integer objects on a stack.

```
public void stackTest() {
    Stack stack = new Stack();
    time {
        for(int I=0;I<100;I++){
            stack.push(new Integer(i));
        }
    }
}
```

A keyword has a number of *parameters* that can be keywords themselves. The time keyword has one parameter named *body* to handle the body content. This is called keyword *instantiation*. The while keyword for example has two parameters: a condition and a body.

```
while (condition) { body }
```

Applying this parameterisation mechanism results in the above, mentioned keyword trees. The example below is a specification of a simple language extension for java. For the time being, XML is used to describe the language; later on a proper DSL will be developed to describe the language to benefit from the power of a DSL. The language is built out of two keywords GCK (general code keyword) and Time. The GCK is a keyword, which definition is not given here, that takes one other keyword as a parameter. The GCK keyword tries to match the other keyword, when that fails it absorbs the current token in the input and retries again; it continues to do so until the end of the input is reached.

```
<?xml version="1.0"?>
<!DOCTYPE compiler SYSTEM "project.dtd">
<compiler>
  <name> TimerJavaExtension </name>
  <path>c:\data\Compilers\TimerJavaExt\</path>
  <package>timerjavaext</package>
```

```

<keywordclasspath>
  <path>c:\data\Compilers\TimerJavaExt\</path>
</keywordclasspath>

<keywordlist>
  <keyword type="keywords.GCK" id="content">
    <param name="body">
      <keyword type="keywords.Time">
        <param name="body">
          <keyword refid="content" />
        </param>
      </keyword>
    </param>
  </keyword>
</keywordlist>

<parameters>
  <parameter name="file" description="..." mode="mandatory" />
</parameters>
</compiler>

```

A keyword can be re-used as such or can be re-instantiated in another context (another place in the keyword tree). Pieces of keyword trees that will be re-used can be put next to the root keyword tree. A keyword instantiation can have an identifier; the identifier of the GCK keyword instantiation is content.

```
<keyword type="keywords.GCK" id="content">
```

When a particular instantiation is reused, its identifier is used to reference it. In the example above, the instantiation of the keyword GCK is referenced in the body parameter of the keyword time.

```
<keyword refid="content" />
```

The components should also be reusable in other DSLs, so we further drain the keywords. When keywords are composed to build a language, they often need information of other components. Since keywords are to be used in different DSLs they cannot obtain this information, as they don't know which other component contains the required information. In the language specification we provide this extra information because it's a super structure of the whole language, where such information is available and known. That is why a language specification acts also as glueware between the keywords.

## 2.2. Linguistic components

Keywords are the basic linguistic components of a DSL. They have 3 parts: a data structure that holds the keyword information, a syntax description and code that implements its effect. Our implementation uses java classes to hold the keyword information and supports the use of java code to implement its effect. The keyword is described in a proper DSL to benefit from the power of DSL to write other DSLs. It abstracts away as much as possible the details of the DSP (domain specific processor) and development environment. The time keyword is described as follows:

```

keyword time {
  AST {
    public class AST {
      keyword body;
    }
  }
  RecognitionPattern {
    ( "time", "{", keyword body, "}" )
  }
  generator {
    public JavaStatements generate() {
      JavaStatements js = generate body;

      >> long aXX = System.out.currentTimeMillis();
      >> $js;
      >> System.out.println("action has taken " +
        (System.currentTimeMillis() - aXX));
    }
  }
}

```

I will only discuss the example above; the DSL has more options, features and extensions to support the development of keywords. The AST (abstract syntax tree) is a java class where the information of a keyword is kept. We extended the java language with a new kind of datamember keyword body. During compiler generation this keyword will be instantiated and a java datamember will be generated that will match the type of the given body keyword parameter. During parsing AST objects are created. In the example the result of the given body keyword parameter parsing is a proper AST and is put into the AST of the time keyword. The recognition pattern describes the syntactic structure of the keyword. It is written in another proper DSL that will transform the description into parse code. The DSL used here is an extended regular expression parser. There are two extensions added: the referential and the generalcode. The referential will try to match a complex regular expression e.g. `id()` will match an identifier of the form `('a'-'z' | 'A'-'Z')+ (('a'-'z' | 'A'-'Z')* | '_' )+`. All referentials are derived from other keywords supplied via the language specification. The generalcode matches every token until parsing can be resumed. The former allows you to separate common recognition patterns and use recursive definitions. The latter is there to support layered and fast creation of DSLs; generalcode is there to absorb every alien token.

Single character	'x'	And	( sub1 , sub2, ... )
Character range	('x' - 'z' ) (15 - 98)	Or	( sub1   sub2   ... )
String	"string"	Optional	!( sub )
Referential	name()	ZeroOrMore	( sub )*
GeneralCode	gc		

The generator we used in this example returns an abstract syntax tree of java statements. Enabling other components to perform target language level transformations. The generator can have several methods, datamembers, etc. In this case there is only one method declaration, which makes it the main generator method. In the first statement, `generate body` will initiate the effect of the body parameter keyword. In this case a `JavaStatement` is returned. The next three lines will be parsed into an abstract syntax tree and returned to the initiator of this generator.

In general a generator can be anything. A generator could just return a piece of generated code; use a code writer framework; or the full abstract syntax tree of the target language. However I would like to stress that a generator for a DSL could also transform a structure into another and return the result, or just check some assertions. Therefore there is no default transformation-guiding engine! Every keyword decides how to operate, how to insert, transform or delete target language code or how to check assertions.

### 3. Related work

How does keyword based programming relate to intentional programming, open compilers, etc. ? In embedded DSLs like embedded SQL the DSL is escaping. Keyword-based programming thus covers such languages. There is an interesting relationship between keywords and instrumentation tools. Instrumentation tools can be built via monitoring keywords. By adding keywords to the language specification they come available to be used inside a DSD. Keywords have an arbitrary syntax description allowing you to work with keywords in basically two ways (in this case):

- Transparently changing the language semantics
- Requiring to intentionally adapt the DSD.

The former is implemented by defining no extra syntax by the keyword. The DSD won't have to be changed and the keyword is used transparently. It will thus be active in all the areas of the DSD covered by that part of the language specification. To achieve the latter, extra syntax is defined in the keyword. The DSD has to be changed in order to activate/use the keyword. It is often a good thing to state explicitly the intention that deals or copes with a certain issue. In the field of distributed computing one has long attempted to deal with the distributiveness of a system in a transparent way for the programmers point of view. Because of the very different issues raised by distributiveness to traditional environments one has come convinced that it should be your clear intention to make your program distributive [Ken94].

Intentional programming argues that raising the abstraction level of programming languages is limited by the syntax rules. Instead of using a text stream to store their program they store the abstract syntax tree composed out of intentions. Each intention can be visualized in a graphical form. Keyword based programming keeps the syntax and thus having the following benefits:

- Familiar system. People and languages go back a long time. In languages we state problems, express ideas and solutions, to communicate our intentions to third parties.
- Related utilities like versioning control, repositories, etc keep on functioning. Many of these tools are text base. The same tools can be used for the enriched code that is produced but also for the keywords and language specification itself.

However the syntactical representation of code is not hard coded in the environment; a graphical user interface for example could relatively easy be put on top of, or next to the current text oriented approach.

JSP Taglibs is a very common, widely used language extension mechanism nowadays. JSP (Java Server Pages) is a technology to build dynamic web pages. It translates a page to servlets, java classes executed upon a client request. The taglibs extend the jsp directives. The word taglib has of course a fixed syntax

```
<name attrib1="value" attrib2="value">
  body
</name>
```

with only one body. The generators can basically do anything but they always return something to their parent via a text stream. Their parents can interpret these via string manipulations. Taglibs are like keywords but with a fixed syntax and thus an AST with as attributes name-value pairs and one keyword parameter. The generators could return a string or stream.

#### 4. Conclusion and future work

We have developed an environment where domain specific languages can be built. DSLs are composed out of linguistic components called keywords. These highly reusable cross-DSL components allow the building of new and the adapting of existing DSLs. Languages are therefore becoming open and customizable tools.

Additional work has to be done in order to open up the development environment itself so the distinction between language components and environment components fades away. Tools are needed to warn, prevent or even fix specification when a DSL will not act or does not act like it should. Keywords must be carefully examined while they evolve to guarantee and further develop their reusability within the same and within other DSLs.

Instead of one recognition pattern, keywords will be extended to support various recognition patterns that can be parameterized according to the language context in which they are used. For example in the time keyword the brackets may make less sense in the Cobol language and could be parameterized so that instead of brackets, BEGIN and END could be used.

The generators are constraining the reuse of a keyword. In the Time keyword, generate body will initiate the effect of the body parameter keyword. The generator expects to return an instance of JavaStatements, if this precondition is not met the incompatibility will be discovered only at compile time of an DSD. This is probably only one example of the static verification issue at hand, which has to be investigated more thoroughly. It is likely that some support for multiple generators for one keyword having different semantic and implementation requirements should be provided in the definition of a keyword.

#### 5. References

- [Bat97] Y. Smaragdakis and D. Batory, DiSTiL: a transformation library for data structures, USENIX Conf. On Domain Specific Languages (DSL 97), 1997.
- [BRJ99] G. Booch, J. Rumbach, And I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, Reading, MA, 1999.
- [Due97] A. Van Duersen and P. Klint, "Little languages: Little Maintenance?", Proc. First ACM SIGPLAN Workshop on Domain-Specific Languages, Paris 1997.
- [Ken94] Samuel C. Kendall, Jim Waldo, Ann Wollrath and Geoff Wyant, A note on distributive computing, Sun Microsystems research, TR-94-29, 1994.
- [Rod99] Roeder, Lutz. Transformation and Visualization of Abstractions using the IP System, Invited Talk given at the GCSE'99 Young Researchers Workshop at the 1st International Symposium on Generative and Component-based Software Engineering, 1999.
- [Sim95] C. Simonyi, The death of computer languages, the birth of Intentional Programming, NATO Science Committee Conference, 1995.
- [SCK+96] M. Simons, D. Creps, C.Klinger, L.Levine, and D. Allemang. Organization Domain Modeling (ODM) Guidebook, Version 2.0. Informal Technical Report for STARS, STARS-VC-A025/001/00, June 14, 1996.
- [WAL96] Domain Specific design languages. URL <http://www.cse.ogi.edu/~walton/dsdls.html>.