Object Oriented Metrics: Precision Tools and Configurable Visualisations

Warwick Irwin and Neville Churcher
Software Visualisation Group, Department of Computer Science,
University of Canterbury, Private Bag 4800,
Christchurch, New Zealand
{wal,neville}@cosc.canterbury.ac.nz

Abstract

Software metrics are a valuable tool in helping software engineers to develop large, complex software systems. However, it is vital that transparency and precision are maintained at all stages. We contend that without grammars we cannot define metrics rigorously, without transparent and powerful parsing tools we cannot collect data accurately and without flexible configurable visualisation we cannot exploit the full potential of our data. In this paper, we report the development of JST, a semantic analyser for Java, and show how it is incorporated into our pipeline-based approach to metrics collection and visualisation. We describe a new visualisation, class clusters, which not only demonstrate the data generated by our tools but also illustrate the value of 3D virtual worlds for visualising software metrics.

Keywords: software visualisation, metric visualisation, static analysis, VRML, XML, software metrics, object-oriented metrics, empirical software engineering

1. Introduction

The advantages of object-oriented (OO) and other modern software engineering techniques are offset by the continuing increases in the size and complexity of software systems. These remain among the greatest challenges faced by software engineers and contribute to the failure of many projects [17]. Projects involving millions of lines of code are no longer exceptional. Such systems are beyond the abilities of individuals to comprehend and their evolution poses further challenges.

The richer semantic models underlying approaches such as OO involve more kinds of components and relationships than their procedural counterparts. Consequently, envisaging the "neighbourhood" of a component (related classes, methods, ...) is often far from simple, particularly when subtle effects such as overloading are taken into account, yet this knowledge is necessary for many design and development activities.

Software engineers have typically addressed these issues by adopting some form of the Goal/Question/Metric (GQM) paradigm [2] whereby direct or indirect measurements allow specific properties of software to be evaluated in order to address specific issues. This process must occur in the context of models which relate the observable metrics to the more abstract quantities, such as reusability, which are ultimately required.

Many software metrics have been proposed [11, 24, 22, 16] in order to represent particular aspects of software. Typically, a number of distinct concepts must be represented so multiple metrics are needed. This leads to large volumes of inter-related data and consequent difficulties in interpreting and acting on it.

Although many advances have been made, several major problems remain. We advocate transparent strong parsing techniques and pipeline-based visualisation as a way to address them. Examples of outstanding issues considered in this paper include the following:

precise definition of the most suitable set of metrics to collect in order to answer particular questions can be remarkably challenging. Even relatively straightforward metrics, such as the number of methods in a class, can be difficult to define precisely [10]. The details are often very significant (e.g. are constructors, inherited, overloaded, overridden, private, ... methods included?) and may be hard to change later, during exploratory analysis. We advocate basing metric definitions strictly on publicly-available grammars, and exposing detail and design decisions through XML formats, in order to enable precision later in the process. Through our visualisations, we attempt to present metrics data in such a way that consistent visual metaphors can help to convey such details.

data integrity has been a long-term concern for empirical software engineering and metrics researchers. In practice, available tools are rarely able to guarantee the correctness, completeness and self-consistency of data (i.e. that it is accurate and measures what we think it does). Sometimes, potentially important data is silently ignored or decisions are not recorded. For example, how do Java's anonymous inner classes and try-catch blocks contribute to the cyclomatic complexity of the methods which contain them? Our use of strong parsing techniques allow instrumentation to record any quantity defined in the grammar. By exposing the intermediate products (which, in turn, are based on grammars) used in metric calculations, we enable others to compare, calibrate and re-compute values if desired. We have chosen to face head-on the problem of parsing potentially ambiguous or otherwise diseased standard grammars rather than accept loss of data quality with its consequences for meaningful metrics analysis and visualisation.

information overload, with corresponding loss of productivity and increase in errors results from the enormous amounts of data generated by collecting multiple metrics on large projects. The effort required to "see the wood in the trees" may be comparable to the effort involved in direct analysis of the source—negating the usefulness of metrics in many cases. We use information and software visualisation techniques not only to present multiple variables but also to provide visual metaphors which assist the user by giving a more "holistic" view of the system.

The essential message we aim to convey in this paper may be summarised as "better parsing for better data; better visualisation for better understanding".

In our recent work, we have developed a pipeline-based approach which encompasses each step from grammar to visualisation. XML is used to provide self-describing data representations at each stage and XSLT transformations are used in many of the transformation steps [3, 35].

The pipeline begins with a grammar for the relevant programming language. This is used to generate an instrumented parser which is used to extract raw parse tree data from source code. At the end of the pipeline, we use sets of mappings to transform the data into whatever form is required for visualisation. We have found the Virtual Reality Modelling Language (VRML) [4] to be particularly effective as a means of communicating information about software structure and encourage the reader to visit our web page [33] to see some examples.

In this paper we describe an important new extension to the pipeline: a semantic analyser (JST) for Java. This exposes semantic information in a Java program, producing a model of an entire, fully cross-referenced program. This component greatly extends the range of quantities which

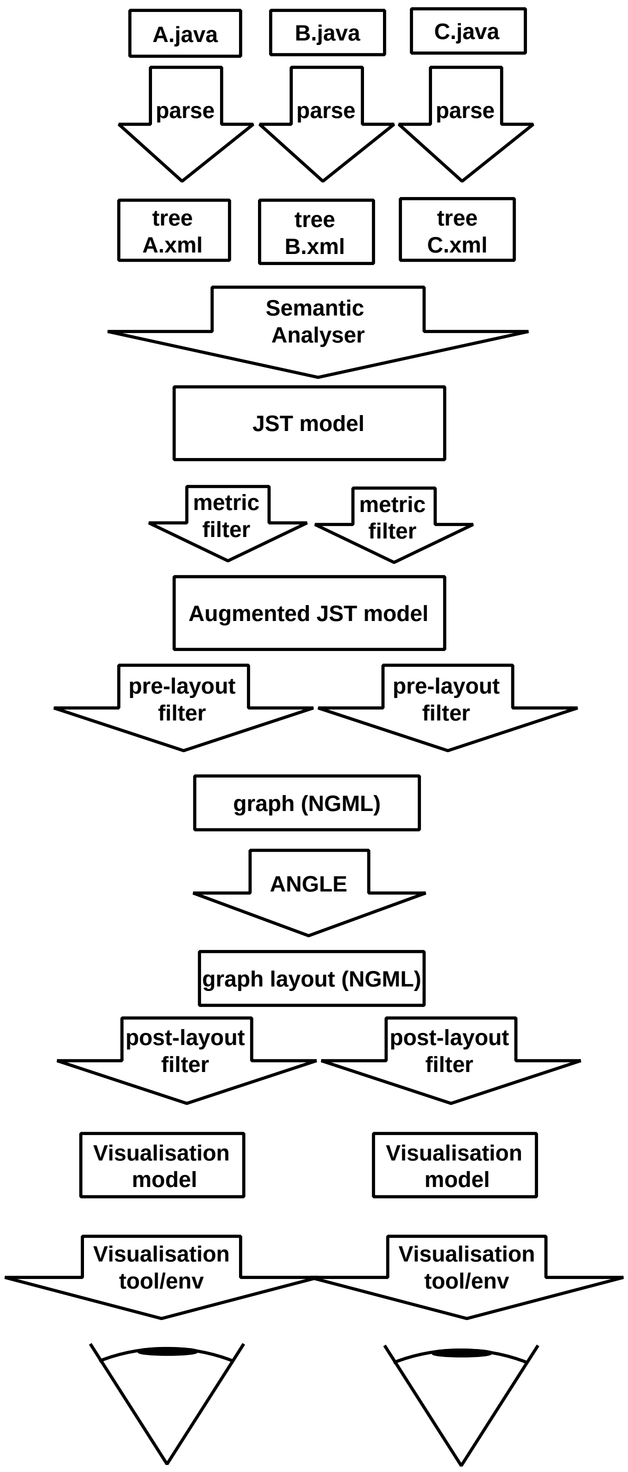


Figure 1. The metrics visualisation pipeline

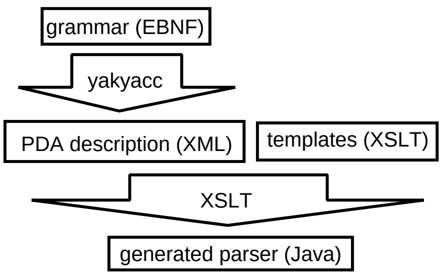


Figure 2. yakyacc **parser generation pipeline**

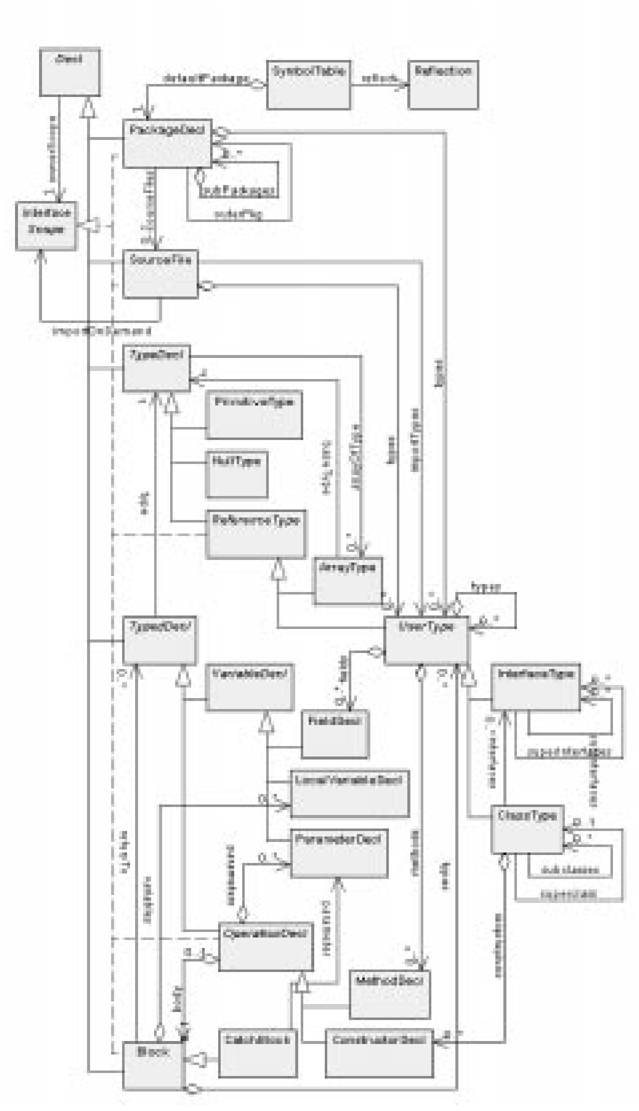


Figure 3. JST meta-model

may be analysed. The model is represented in XML and fits naturally into the existing pipeline.

We present a new visualisation, the class cluster, (which would not have been possible without JST) of some important relationships between classes and illustrate how metrics derived using JST enrich the visualisation as well as easing its interpretation.

The remainder of the paper is structured as follows. In the next section we describe our pipeline approach. We introduce JST in Section 3 and discuss the critical rôle of parsing technology in obtaining reliable data. In Section 4 we discuss the visualisation of metrics and the rôle of mappings. Class clusters are developed in Section 5 and an example of the pipeline process is given. Our conclusions and plans for further work are presented in Section 6.

2. The metrics visualisation pipeline

Figure 1 shows a typical metrics visualisation pipeline used in our work. It consists of a sequence of filters for parsing source code, augmenting it with metrics and transforming it into visualisations.

The first step is parsing. This makes the syntactic structure of source files explicit, by adding to the source text XML tags that describe the parse tree. The second step is semantic analysis using JST. The semantic analyser reads the parse trees and makes semantic information explicit. Unlike parse trees, which can be directly imposed upon source code, the semantic model is independent of lexical structure and so is described in a separate XML sub-tree. Relationships between the semantic model and parse trees are captured in XML.

Parsing and parser generation are discussed further in Section 3 and examples of the XML representation of the products at each pipeline stage are given in Figure 4.

Pipelines need not be linear: they may branch, merge and even contain cycles. This contributes significantly to the flexibility of the architecture, and is particularly helpful for experimenting with metrics and visualisations. A given intermediate file may be transformed in a variety of ways with little extra effort, making the approach highly suitable for exploratory activities.

Different programming languages are accommodated by generating a parser specific to each and developing appropriate downstream transformations. Parser generation is achieved using the yakyacc tool described elsewhere [26]. The parser generation process, which produces the parsers used in the first stage of Figure 1, is illustrated in Figure 2.

The use of XML for intermediate files makes their structure explicit: the files contain their own metadata. This makes the pipeline much more transparent than would be the case for implied metadata. More importantly, the embedding of metadata in the files decouples filter programs from each other by externalising file formats. Exposing metadata in this way enables others to make meaningful comparison with our results.

Filters are used at several stages in the pipeline to select and transform data for use in subsequent steps. Transformation of XML is well supported by existing tools, particularly XSLT [28, 35] which employs a powerful stylesheet language to transform the data into a new format (usually another XML file in our application). XML processing support (such as DOM and SAX parsers) is available for a range of scripting and programming languages and these may be used to write filters if desired.

Once the syntactic and semantic structure of a program has been made explicit, it is relatively straightforward to derive diverse metrics and models of the software. Using the later stages of the XML pipeline, we can transform the JST model into conceptual and visual models by selecting and combining the features of interest as described in Sections 4 and 5.

3. Parsing, semantic analysis and JST

Parsing of source code is a necessary precursor to producing metrics and visualisations of software structure. Detailed information on grammars and parsing is available in standard texts [1, 20, for example]. In earlier papers [25, 26, 7] we argued the importance of a strong transparent parsing approach when parse information is to be used in metrics and visualisations. Metrics that convey syntactic information ought to be defined in terms of a standard definition of a programming language.

In the absence of a rigorous and detailed specification mechanism, even relatively straightforward metrics, such as the number of statements in a method or the number of methods in a class, are difficult to define, capture and interpret [10]. Even widely-used metrics, such as WMC and LCOM from the Chidamber & Kemerer suite [5] have generated confusion and debate [9, 22, for example]. Grammars provide an essential reference point for defining metrics, calibrating tools, resolving inconsistencies, and enabling reproducibility by comparing results from different researchers.

Frequently, standard grammars for programming languages are designed to aid communication of the concepts of the language, rather than to yield to conventional parsing approaches such as LL(k) and LALR(1). As a result, the grammar may contain ambiguities or other undesirable properties.

Consequently, as we have discussed elsewhere [25], parsers used in practice are often generated from non-standard grammars adapted from the standard grammars in order to conform to the constraints of a given parsing approach. Such modified grammars significantly complicate the definition and production of metrics. Either a metric must be defined in terms of the modified grammar (in which case the modified grammar must be understood by anyone interpreting the metric's values), or the parse information must be translated to a form consistent with the standard grammar before the metric is calculated.

Difficulties such as these can be avoided if a stronger parsing approach capable of recognising the standard grammar is used. We have found that Tomita (GLR) parsing [34] is well suited for this purpose. It can parse complex ambiguous programming languages (including C^{++}) using the standard grammar, in near-linear time.

Tomita parsing has the further advantage of simplifying parsing that relies on semantic disambiguation. Weaker parsing techniques do not tolerate syntactic ambiguity, so parsing must be closely coupled to semantic analysis; the semantic information is used to eliminate ambiguous syntax during the parse. In some languages, notably C^{++}, this coupling greatly complicates both the parsing and the semantic analysis [31]. With a Tomita parser, syntactic ambiguity is tolerated during parsing and subsequently may be resolved by a semantic analyser. Consequently, parsing and semantic analysis are decoupled, and semantic analysis is simplified by the explicit identification of all syntactic ambiguities.

Our yakyacc tool (see Figure 2), which generates the parsers used in the pipeline, will generate a Tomita parser if the grammar contains ambiguities.

3.1. Semantic analysis and JST

The separation of parsing and semantic analysis fits naturally with our pipeline approach.

A parse tree represents the syntactic structure of a source file. The syntactic structure of an individual file contains sufficient information for metrics and visualisations of many aspects of program structure, including some representations of class cohesion [7]. However, metrics and visualisations based on raw parse trees are restricted to features that can be discerned from syntax alone.

In a parse tree, semantic information is not evident. So, for example, separate branches of a tree represent a method invocation expression and the declaration of the method which is being invoked. If, as is likely to be the case, an invocation occurs in a different source file from a declaration, the branches will be in separate parse trees. Consequently, metrics and visualisations based on syntactic information alone cannot include method invocations, or other semantic connections such as field (attribute) accesses. This is a severe restriction for metrics since, without such connections, complete and accurate measurements of metrics such as module coupling cannot be made.

Many of the connections between the structural elements of a program are discernable only at a semantic level. Semantic analysis identifies the structural elements in parse trees, and exposes their interconnections by finding semantic relationships between branches of parse trees. Once all semantic connections have been found, a complete model of the static structure of a program is available.

The essential data structure of a semantic analyser is a symbol table, which represents the semantic features of a program and allows them to be looked up by name. Using a symbol table, a reference in one branch of a parse tree can be resolved by finding the appropriate definition in the symbol table.

We have developed an experimental tool, JST, that reads XML parse trees (as produced by our Java parser [25, 26]), constructs definitions in a symbol table, and looks them up to resolve references. Once all source files have been processed this way, an XML file is produced. It contains annotated versions of the original parse trees, and symbol table information that exposes all the structural elements and connections between them.

The resulting XML file describes the semantically cross-referenced syntactic structure of a Java program. It may subsequently be filtered and transformed to produce a wide range of visualisations and metrics, including characterisations of coupling and reuse.

3.2. Developing A Java symbol table

Our primary motivation for developing JST was to be able to obtain correct, complete and consistent values for arbitrary metrics. For an object-oriented language such as Java, designing a symbol table is not a trivial task. If it is to correctly resolve all references, it must conform to the language standard, including the scope rules and naming practices of features such as packages, superclasses, inner classes, interfaces, static members, and most significantly, overloaded methods.

A small number of public domain Java symbol tables are available. One, which was part of a simple cross-reference tool for Java 1.1 [32], was subsequently developed into javasrc, an open-source hypertext cross-referencer for Java 1.3 [27]. Unlike JST, these tools do not attempt to resolve overloaded method calls, and have several other limitations, including simplified package naming, some syntax limitations, and no handling of anonymous classes or member access specifiers.

We investigated improving code from javasrc as the basis for our symbol table, replacing its ANTLR [29]-generated parser with a simple reader of our XML parse trees. However we ultimately chose to design our own classes to more closely reflect the concepts described in the Java Language Standard [19]. This not only allowed us to be more confident that our code conforms to the standard but also gave us a good structure on which to graft the missing—but essential— features, including overloaded method resolution.

An alternative approach, deriving symbol table information from `.class` files using Java's reflection API, was also considered. We are interested only in analysing syntactically correct source code, and such code will usually have a corresponding `.class` file emitted by javac (or some other Java compiler). Java compilers embed symbol table information in `.class` files so that this information may be reported by the reflection API. Typically, Java programs use reflection to dynamically load classes that were unavailable at compilation time.

We investigated using reflection to extract symbol table information, and then using that information to connect semantically related portions of our parse trees. The apparent appeal of this approach was threefold:

- All of the symbol table information was available in advance, so it reduced the complexity added by order dependencies between declarations and look-ups as they were discovered in the parse trees (e.g. because properties may be used before they are declared).
- Symbol table information was available for classes for which source code was not present, including standard library classes such as java.lang.Object or COTS components.
- Third-party support for resolving overloaded method calls was potentially available.

The reflection API can look up a method, given its name and parameter types. This is sufficient data for the API to perform method resolution, but as other authors have noted [23], it does not. The reflection API performs only an exact match on parameter types, whereas full method resolution must consider type promotions and find the most specific method from a set of applicable methods. Holser [23] provides a BetterMethodFinder class that extends the reflection API to include proper method resolution.

This reflection-based method resolution proved to be less helpful to us than anticipated. The reflection API is designed to expose the *public* interface of a class, and the method resolution is accordingly suitable for client classes that are not part of the protected, private or default (package) scopes. This restriction, while appropriate for reflection's intended purpose, does not necessarily hold for the classes we are analysing; we need to resolve calls of all methods, not just public ones. In addition, we are interested in a class' internal use of methods, fields, local variables, and parameters. This information is not available through reflection but is necessary in order to evaluate many metrics of interest.

We concluded that reflection did not eliminate the need to build our own complete symbol table from parse tree information, but was useful for resolving references to classes for which we did not have source code. This means that the symbol table is populated with declarations found in parse trees, and whenever a look-up references some external symbol, such as java.lang.String, the missing information is supplied by reflection.

3.3. JST

The resulting tool, JST, runs once per program, (much like a conventional linker). It reads parse trees into memory, and then walks through them to discover declarations and references. This information is stored in the symbol table; each declaration retains links to parse tree portions that reference it. Any references that are not resolved by declarations found in the source code are supplied by reflection. Finally, the complete structure is written out as an XML file. Figure 1 shows JST's placement in a typical pipeline.

The symbols in the symbol table represent components of a Java program such as packages, classes, interfaces, fields and methods. Together, these objects form a semantic model of the program. Figure 3 shows a UML class diagram of the meta-model. Although it appears complex at first glance, most classes represent concepts with which every Java programmer is familiar. For example, every class declaration is represented by an instance of ClassType, and every method declaration by an instance of MethodDecl. Inheritance is used wherever properties can be generalised.

The purpose of most classes in Figure 3 is self-evident, and need not be discussed in detail here. The following list describes some of the less obvious features of the model:

- Class SymbolTable is the main entry point to the semantic model. It is a lightweight class (for a symbol table) that delegates model representation and lookup to the declaration classes. Classes unavailable in source code are referred to the Reflection class.

- Class Decl is the root of the declaration hierarchy. Packages, classes, fields, methods and parameters each inherit from Decl. Each declaration has a name and occurs in some scope. Names are generated for anonymous declarations. The concept of a declaration has been loosened to encompass all relevant semantic features in a Java program. Specifically, source files and blocks (sequences of statements delimited by braces) are also considered to be declarations. This simplifies the design by allowing all program features to be treated consistently at an abstract level.

- Scope is an interface that declares methods for looking up declarations by name. Any declaration that can contain other declarations is a Scope, and provides lookup methods to retrieve its contents by name. Thus, declarations implement the scope and lookup rules of the language; each type of declaration knows its own structure and rules for looking up names. If a lookup fails within a declaration, the request is passed up to the declaration's owner scope. In this way lookups search progressively wider scopes without the need for a current scope stack.

- SourceFiles are explicitly represented in the model. This is helpful because source files provide lexical scope and participate in the lookup of classes by importing packages and classes.

The data needed in order to evaluate arbitrary code metrics is now available: the semantic model allows it to be delivered in a form suitable for analysis.

3.4. Building the semantic model

Each source file in a Java program is syntactically complete (i.e. is a translation unit), but usually has semantic dependencies on other source files. For example, a class typically inherits from, and uses methods of, classes in other source (or `.class`) files. In order to make all semantic dependencies explicit, the semantic model must span an entire program. The current version of JST builds the semantic model monolithically: all parse trees for a program are loaded and the complete model is assembled before being saved as an XML file. This approach keeps implementation relatively simple, though its memory requirements are higher. However, it has the advantage that metrics and semantic data are evaluated in the context of a well-defined snapshot of the classes constituting the program. This is particularly important for Java, where CLASSPATH settings can determine at run-time precisely which classes are loaded.

Many semantic checks in Java must occur in an order different from the syntactic structure of a file. For example, attributes may be used in a source file before they are declared. More fundamentally, Java (unlike C++) does not distinguish the declaration of a feature from its definition, so the syntactic structure does not guarantee that features will be declared before they are used. A semantic analyser for Java might choose to remember unresolved references until the target declarations are discovered, or alternatively, might process the information in parse trees in an order that ensures declarations occur before usages. JST takes the latter approach.

JST constructs the semantic model in a series of steps. The order in which parse trees are processed ensures that every feature (package, class, method, parameter, . . .) of a program is declared before it is looked up. This means that when a method invocation lookup occurs, the class inheritance hierarchy is known fully and can be searched to find

```
// 19.8.3) Method Declarations
method_declaration :
      method_header method_body
   ;
method_header :
      modifiers_opt type method_declarator throws_opt
   |  modifiers_opt VOID method_declarator throws_opt
   ;
method_declarator :
      IDENTIFIER LPAREN formal_parameter_list_opt RPAREN
   |  method_declarator LBRACK RBRACK // deprecated
   ;
```

(a) **Grammar productions**

```
public abstract class TypedDecl extends Decl {
   protected TypeDecl type;
   public TypedDecl(Scope theOwner,
         String theSimpleName,
        Nonterminal theSource) {
     super(theOwner,
        theSimpleName,
        theSource);
    }
   // ...
}
```

(b) **Java code**

```
<method_header>
 <modifiers_opt>
  <modifiers>
   <modifier>
    <token id='PUBLIC'>public</token>
    </modifier>
   </modifiers>
  </modifiers_opt>
 <type>
  <reference_type>
   <class_or_interface_type>
    <name>
     <simple_name>
      <token id='IDENTIFIER'>TypeDecl</token>
     </simple_name>
    </name>
   </class_or_interface_type>
  </reference_type>
 </type>
 <method_declarator>
  <token id='IDENTIFIER'>getType</token>
  <token id='LPAREN'>(</token>
  <formal_parameter_list_opt>
  </formal_parameter_list_opt>
  <token id='RPAREN'>)</token>
 </method_declarator>
 <throws_opt>
 </throws_opt>
</method_header>
```

(c) **Parse tree**

```
<method id='MTH_jst.symtab.TypedDecl.getType()' name='getType()'
     source='NTL_53719'
     type='TYP_jst.symtab.TypeDecl' modifier='public'>
<block id='BLK_jst.symtab.TypedDecl.getType().@BODY'>
<reference to='FLD_jst.symtab.TypedDecl.type' from='NTL_53694'/>
</block>
</method>
...
<nonterminal id='NTL_53719' type='method_declaration'>
 <nonterminal id='NTL_53691' type='method_header'>
  <nonterminal id='NTL_53682' type='modifiers_opt'>
   <nonterminal id='NTL_53681' type='modifiers'>
    <nonterminal id='NTL_53680' type='modifier'>
     <terminal type='PUBLIC'>public</terminal>
    </nonterminal>
   </nonterminal>
  </nonterminal>
  <nonterminal id='NTL_53687' type='type'>
   <nonterminal id='NTL_53686' type='reference_type'>
    <nonterminal id='NTL_53685' type='class_or_interface_type'>
     <nonterminal id='NTL_53684' type='name'>
      <nonterminal id='NTL_53683' type='simple_name'>
       <terminal type='IDENTIFIER'>TypeDecl</terminal>
      </nonterminal>
     </nonterminal>
    </nonterminal>
   </nonterminal>
  </nonterminal>
 <nonterminal id='NTL_53689' type='method_declarator'>
  <terminal type='IDENTIFIER'>getType</terminal>
  <terminal type='LPAREN'>(</terminal>
  <nonterminal id='NTL_53688' type='formal_parameter_list_opt'>
  </nonterminal>
  <terminal type='RPAREN'>)</terminal>
 </nonterminal>
 <nonterminal id='NTL_53690' type='throws_opt'>
 </nonterminal>
</nonterminal>
<nonterminal id='NTL_53718' type='method_body'>
...
```

(d) **JST model**

Figure 4. Representation of metrics visualisation pipeline intermediate products

all in-scope methods of the given name. The resulting set of candidate methods is then filtered to find a single method that is accessible, and has parameters that are applicable and more specific than those of other candidate matches (as described in the Java language standard). This process is essential in order to obtain accurate values for metrics such as RFC [5] which involve method invocation.

Initially, all parse trees are examined for declarations; every declared feature in a nameable scope is instantiated in the model. These declaration objects are, at this stage, related only by their lexical containment structure (e.g. an inner class is related to its outer class, which in turn is related to its source file. Each declaration is given a name, but no names are looked up yet.

To complicate matters, some declarations (such as attributes) have semantic scope while others (such as local variables and classes declared inside blocks) have lexical scope: They are accessible only from the point of declaration to the next closing brace. Lexically scoped declarations are omitted from the initial population of the model, as they should not be found by lookups until the appropriate point in the parse tree is reached.

Once all initial declarations are known, they are connected together by looking up the names of features they reference. This cross-referencing happens in the following order:

- Packages and classes named in import statements are found. These must be known before classes can be looked up.
- Superclasses and interfaces named in `extends` and `implements` clauses are found. This inheritance structure is needed for subsequent lookups.
- The types of all class members (attributes and methods) are found. All types must be known in order to analyse expressions such as `a.b.c.d` since the type of `a` must be known to determine the existence and type of `b`, and so on.
- Each code block is processed statement-by-statement, in parse tree order. Lexically scoped types and variables are instantiated. Identifiers in expressions are looked up. At any point during cross-referencing, a lookup of a named type (class or interface) will fail if a parse tree for that class or interface was not provided to JST. Whenever this happens, Java's reflection API is used to load public interface information from .class files so that the semantic model is complete and all references are resolved.

At the end of this processing, all connections between declarations have been recorded and the semantic model is complete. It is available as an XML file that may be transformed to produce metrics and visualisations.

3.5. XML representation

The representation of some of the pipeline stages discussed in this section is illustrated in Figure 4. Figure 4(a) shows part of the EBNF representation of the Java grammar which is used in the generation of the parser by yakyacc.

Figure 4(b) shows a Java source code fragment (from `TypedDecl.java`), the input to the first stage of the pipeline shown in Figure 1, while Figure 4(c) shows the part of the corresponding parse tree (`TypedDecl.xml`) corresponding to the phrase `public TypeDecl getType()`, which is a `method_header`.

Figure 4(d) shows two parts of the semantic model produced by JST from `TypedDecl.xml`. The first part shows the semantic model of `TypeDecl.getType()`, and the second shows the syntax tree as it appears in the same XML file, including the cross-references to other parse trees.

XML's text-based, self-describing formats provide ready transparent access to this data.

4. Visualisation

The interpretation of software metrics data poses many challenges apart form the sheer volume of data. There are typically many variables; outliers occur and are likely to be significant; data distributions are skewed and have large ranges.

In our previous work [8, 21, 6, 26, 7], we have demonstrated the advantages, such as the ability to represent more information and to facilitate exploratory analysis, of 3D virtual worlds in software visualisation with application to concepts such as class cohesion. We have used VRML as a platform-independent means of deploying our visualisations to the desktop. Figure 5 shows a browser with a VRML plug-in displaying a 3D VRML world containing a class cluster visualisation (class clusters are described further in Section 5).

Some researchers have considered 2D visualisation of metrics [12, for example] and some initial attempts have been made to explore 3D visualisations of 2D UML notations [18, 30, 14]. Strengths of our approach include its emphasis on the entire pipeline and its provision of a solid flexible basis for extension to a wide range of metrics visualisations.

Ultimately, we are aiming to use metrics to understand relatively abstract properties of the system such as abstraction, encapsulation, inheritance, associations and communication via messages.

This requires an underlying meta-model of software and metaphors for representing it. The meta-model describes the relevant components (classes, methods, . . .) and connections (inheritance, invocation, . . .). Common metaphors

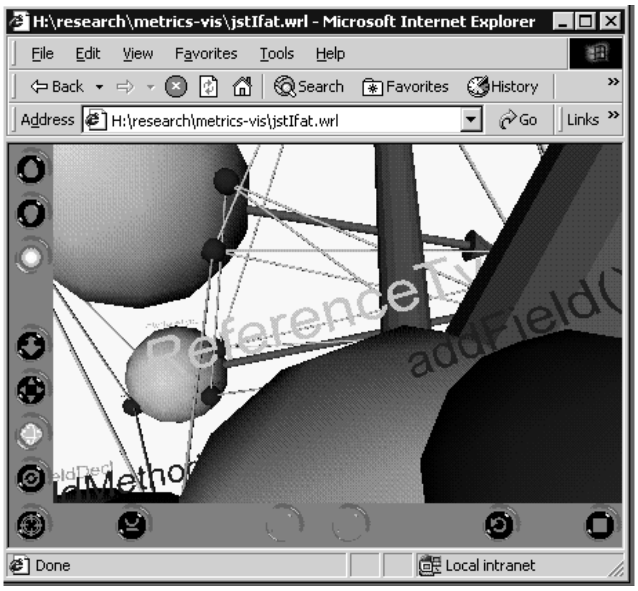

H:\research\metrics-vis\jstIfat.wrl - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address H:\research\metrics-vis\jstIfat.wrl
Go Links
ReferenceTy
addField(
ldMetho
Done
Local intranet

Figure 5. Class cluster world in VRML browser

```
<node type="class" label="TypedDecl" size="2">
  <name>TYP_jst_symtab_TypedDecl</name>
</node>
<node type="method" label="getType()">
  <name>MTH_jst_symtab_TypedDecl_getType__</name>
</node>
<edge type="contains" stiffness="2" l0="2">
  <from>TYP_jst_symtab_TypedDecl</from>
  <to>MTH_jst_symtab_TypedDecl_getType__</to>
</edge>
<edge type="invokes" stiffness=".02">
  <from>MTH_jst_symtab_MethodDecl_getSignature__</from>
  <to>MTH_jst_symtab_TypedDecl_getType__</to>
</edge>
```

(a) **NGML graph representation**

```
DEF MTH_jst_symtab_TypedDecl_getType__
  LabelledMethodNode {
    label "getType()"
    location 21.362E0 5.2825E0 36.967E0
    size 5
}
Edge {color 1 1 0 scale 0.1 0.1
  backBone [28.876E0 36.852E0 45.448E0,
  21.362E0 5.2825E0 36.967E0]
}
```

(b) **VRML**

Figure 6. Pipeline products for visualisation

emphasize particular aspects of the meta-model: examples include UML sequence diagrams (message passing) and class diagrams (attributes, methods and associations).

Our long-term goals include the development of richer metaphors and their realisation through 3D virtual worlds. We begin by exploring applications of the metrics available from our pipeline.

As indicated in Figure 1, the visualisation process takes as input a JST model augmented with metric data and results in the production of specific visualisations. There are three major steps in this process:

pre-filters determine the *content* of the visualisation. Data is selected and transformed into a graph model represented in NGML, an XML-based format. For example, methods might be mapped to graph nodes with attribute values `type="method"` while their cyclomatic complexity values might be mapped to other attribute values. Similarly, inheritance relationships might be mapped to edges connecting the appropriate nodes.

ANGLE [6], computes a 3D layout of the graph constructed by the pre-filters. It uses a force-based approach [13] in which nodes repel each other as if they were similarly charged particles while edges act like springs which pull connected nodes towards each other.

post-filters determine the *look* of the visualisation. Factors such as geometry, colours and visibility are determined in this step. For example, post-filters determine whether public methods will be denoted by shape, transparency, size or colour. Output is typically a VRML file or an XML file destined for further processing before being displayed.

The pre- and post-filter mappings are independent. This important feature allows the appearance of a visualisation to be decoupled from its content. Filters are typically implemented as XSLT stylesheets.

Figure 6 illustrates the representation of the products at this stage. Figure 6(a) shows part of the NGML graph representation involving the `getType()` method whose earlier pipeline representations were shown in Figure 4, while Figure 6(b) shows VRML code specifying the corresponding geometry as it appears in Figure 7.

5. Class clusters

To illustrate our approach, we derive a new visualisation, which we call a *class cluster*. Class clusters highlight aspects of the "neighbourhood" of individual classes (those related by inheritance or method invocation and hence likely to be considered together during development)

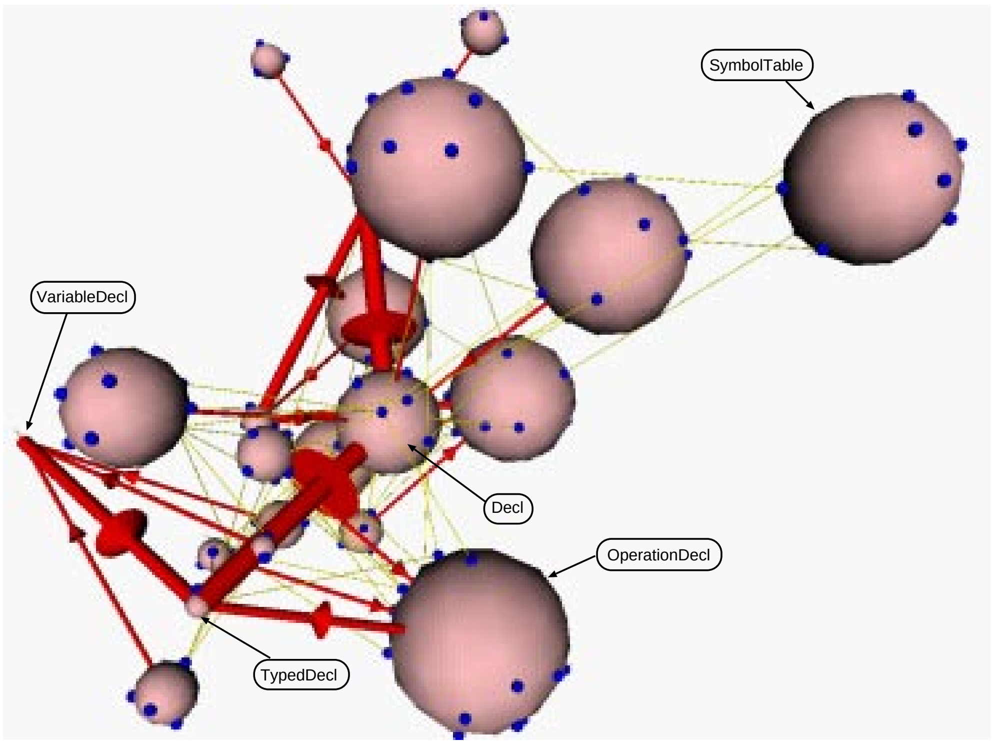


Figure 7. Class cluster visualisation of JST's jst.symtab **package (see also Figure 3)**

while also showing large-scale structure of a system (a single package in our example). The idea is that strongly-related classes should be physically close to each other in a manner reminiscent of statistical clustering [15]. Conventional layout algorithms (such as those commonly employed by IDEs and diagramming tools) generally do not have this property.

Figure 7 shows a snapshot of a 3D VRML world containing a class cluster representing several aspects of the jst.symtab package which is part of the JST source code. Some classes in the snapshot have been labelled for clarity. Figure 3 is a 2D UML diagram containing the same classes. Static greyscale images do not convey the full impression of these worlds—please visit our web page [33] to experience the interactive colour versions for yourself.

The visualisation is based on a set of mappings which map elements of the JST model to the nodes and edges of a graph. The nodes of the graph represent 23 classes and 153 methods of the jst.symtab package. The edges of the graph represent relationships between the elements mapped to nodes: 19 inheritance connections, 153 class-has-method occurrences and 81 method invocations. This set of mappings corresponds to the pre-layout filter stage in the pipeline shown in Figure 1.

A range of parameters is available to configure layout algorithms used by ANGLE. Those chosen here correspond to class-has-method edges which are very stiff and whose length is proportional to the number of methods in the class. Both inheritance and invocation edges prefer to be somewhat longer and are more easily stretched or compressed, with the invocation edges being the more flexible. This allows the overall structure to take on the shape of interconnected spheres whose relative placement is primarily determined by their inheritance relationships.

In this example, the pre-filter mappings omit many possible elements and connections—such as those representing aggregation relationships and visibility. Hence these do not contribute to the layout process. No visualisation can effectively convey all relevant variables simultaneously. Our approach enables the user to select and modify mappings (including metrics) at will and hence greatly enhances its usefulness in exploratory analysis.

Features to note include:

- Inheritance edges are denoted by (red) edges with arrows pointing to the superclasses. In Figure 7, the thickness of inheritance edges indicates the *number of leaf classes supported* (NLCS) metric.
- Each class is represented by a (red) sphere whose radius is proportional to the number of methods (public, locally-defined, excluding constructors) in the class.

- The sphere representing each class node is studded with smaller (blue) spheres representing its methods. The edges which attach these to the corresponding class nodes are not visible in Figure 7 because they lie within class spheres which the post-filter mappings used did not make transparent.

- Invocation of methods is indicated by the thinner (yellow) lines connecting methods. The edge direction (caller to callee) is not shown but this information is available in the JST model.

The layout algorithm has placed the class, Decl, which is the root of the inheritance structure near the centre of the world. This contrasts with the conventional 2D drawing convention of placing child classes below their parents. However, the worlds produced by this set of mappings place leaf classes closer to the periphery of the (roughly) spherical structure—an arrangement which emphasises the availability of leaf class services to potential clients. The mapping of NLCS to inheritance edge thickness serves to highlight characteristics of inheritance structure.

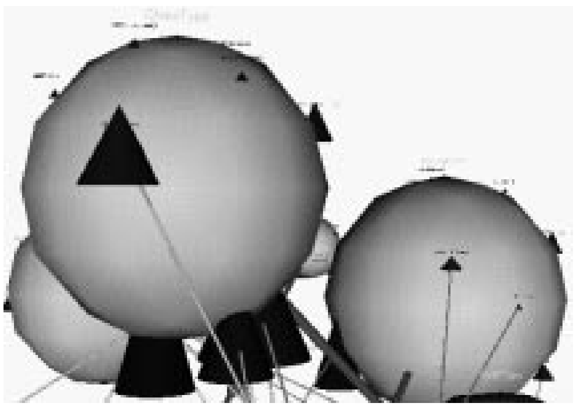
SymbolTable has no inheritance relationships so is located close to classes its methods call (or are called by).

Some classes, such as VariableDecl and TypedDecl, are represented by small spheres because they have few methods (apart from constructors) but nevertheless play important rôles in the inheritance structure. The mapping of NLCS to inheritance edge thickness draws attention for such classes.

Classes which have invocations of parent methods have both inheritance and invocation links. The method of OperationDecl at the "11 o' clock" position is invoking a method of its parent TypedDecl.

By selecting post-filter mappings, it is possible to generate many variations. The worlds shown in Figure 8 illustrate two ways of adding the method level metrics cyclomatic complexity (ν) and number of statements (STMT) to the class cluster visualisation. Figure 8(a) results from mappings which represent methods by cones whose base radii represents ν and whose heights represent STMT values. The user can identify outliers by their aspect ratios. In Figure 8(b), the mappings used include more specific criteria: cubes denote methods where $\frac{STMT}{\nu} \geq 2$ and darker colour denotes lower ν values.

Similarly, visualisations other than class clusters may be derived from the same JST model simply by selecting appropriate filters.


(a)

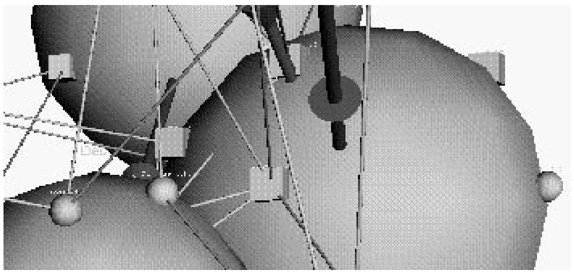

(b)

Figure 8. Inclusion of method metrics

6. Conclusions

In this paper, we have made two major points. Firstly, it is not only possible to derive robust parsers and metrics generators from standard grammars for real languages but it is necessary to do so in order to address issues which arise in understanding software. The addition of JST to our pipeline-based approach essentially removes the limits on the range of static metrics we can compute.

Secondly, having achieved our goal of capturing complete, correct and consistent metrics data through the use of stronger parsing techniques, we demonstrated that 3D visualisation is a valuable tool to help software engineers understand and explore large and complex metrics data sets. Our pipeline approach uses a flexible system of mappings to determine not only which data contribute to the visualisation but also their visual representations.

We hope our tools will permit implicit extension of the GQM paradigm to include visualisation as an integral part.

Exactly how software should be visualised remains an open question, but our approach provides a strong basis for experimentation. Our work in progress includes experimentation with a range of metrics and visualisation metaphors. Anecdotal evidence suggests that our visualisations are useful in practice—however, we intend to conduct user trials to attempt to quantify the benefits of 3D worlds as well as particular visualisations.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.

[2] V. Basili and D. Rombach. The TAME project: Towards improvement-orientated software environments. *IEEE Trans. Softw. Eng.*, 14(6):758–773, 1988.

[3] M. Birbeck, J. Diamond, J. Duckett, O. Gudmundsson, P. Kobak, E. Lenz, S. Livingstone, D. Marcus, S. Mohr, J. Pinnock, K. Visco, A. Watt, K. Williams, Z. Zaev, and N. Ozu. *Professional XML*. Wrox Press, 2nd edition, 2001.

[4] R. Carey and G. Bell. *The Annotated VRML 2.0 Reference manual*. Addison-Wesley, 1997.

[5] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[6] N. Churcher and A. Creek. Building virtual worlds with the big-bang model. In P. Eades and T. Pattison, editors, *Information Visualisation 2001*, volume 9 of *Conferences in Research and Practice in Information Technology*, Sydney, Australia, Dec. 2001. ACS.

[7] N. Churcher, W. Irwin, and R. Kriz. Visualising class cohesion with virtual worlds. In T. Pattison and B. Thomas, editors, *Australasian Symposium on Information Visualisation*, volume 24 of *Conferences in Research and Practice in Information Technology*, pages 89–98, Adelaide, Australia, Feb. 2003.

[8] N. Churcher, L. Keown, and W. Irwin. Virtual worlds for software visualisation. In A. Quigley, editor, *SoftVis99 Software Visualisation Workshop*, pages 9–16, University of Technology, Sydney, Australia, Dec. 1999.

[9] N. Churcher and M. Shepperd. Comment on “a metrics suite for object oriented design”. *IEEE Trans. Softw. Eng.*, 21(3):263–265, Mar. 1995.

[10] N. Churcher and M. Shepperd. Towards a conceptual framework for OO software metrics. *ACM SIGSOFT Software Engineering Notes*, 20(2):69–75, Apr. 1995.

[11] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. Benjamin Cummings, 1986.

[12] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering approach combining metrics and program visualisation. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proc. WCRE’99 (6th Working Conference on Reverse Engineering)*, Atlanta, GA, Oct. 1999. IEEE Press.

[13] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[14] T. Dwyer. Three dimensional UML using force directed layout. In P. Eades and T. Pattison, editors, *Information Visualisation 2001*, volume 9 of *Conferences in Research and Practice in Information Technology*, pages 77–86, Sydney, Australia, Dec. 2001. ACS.

[15] B. Everitt. *Cluster Analysis*. Edward Arnold, 3rd edition, 1993.

[16] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. International Thompson Computer Press, 2nd edition, 1997.

[17] S. Flowers. *Software Failure: Management Failure—Amazing Stories and Cautionary tales*. J. Wiley & Sons, 1996.

[18] M. Gogolla, O. Radfelder, and M. Richters. Towards three-dimensional representation and animation of uml diagrams. In R. France and B. Rumpe, editors, *UML’99 The Unified Modelling Language—Beyond the Standard*, volume 1723 of *Lecture Notes in Computer Science*, pages 489–502, Fort Collins, Colorado, USA, Oct. 1999.

[19] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Palo Alto, CA, 2nd edition, 2000.

[20] D. Grune and C. J. H. Jacobs. *Parsing techniques : a practical guide*. Ellis Horwood series in computers and their applications ;. Ellis Horwood, New York, 1990.

[21] D. Hartley, N. Churcher, and G. Albertson. Virtual worlds for web site visualisation. In *Proc APSEC2000, 7th Asia Pacific Software Engineering Conference*, pages 448–455, Singapore, Dec. 2000. IEEE Press.

[22] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1996.

[23] P. Holser. Limitations of reflective method lookup. *Java Report*, 6(8), Aug. 2001.

[24] D. Ince and M. Shepperd. *The Derivation and Validation of Software Metrics*. Oxford University Press, 1992.

[25] W. Irwin and N. Churcher. A generated parser of C++. *N.Z. Journal of Computing*, 8(3):26–37, June 2001.

[26] W. Irwin and N. Churcher. XML in the visualisation pipeline. In D. D. Feng, J. Jin, P. Eades, and H. Yan, editors, *Visualisation 2001*, volume 11 of *Conferences in Research and Practice in Information Technology*, pages 59–68, Sydney, Australia, Apr. 2002. ACS. Selected papers from 2001 Pan-Sydney Workshop on Visual Information Processing.

[27] K. Johnston. Javasrc. `http://javasrc.sourceforge.net/`, 2002.

[28] M. Kay. *XSLT Programmer’s Reference*. Wrox, 2nd edition, 2001.

[29] T. Parr. Practical computer language recognition and translation: A guide for building source-to-source translators with antlr and java. `http://www.antlr.org/book/`, 1999.

[30] O. Radfelder and M. Gogolla. On better understanding UML diagrams through interactive three-dimensional visualization and animation. In V. D. Gesu, S. Levialdi, and L. Tarantino, editors, *Proc. Advanced Visual Interfaces (AVI’2000)*, pages 292–295. ACM Press, New York, 2000.

[31] J. Roskind. A yaccable C++ 2.1 grammar and resulting ambiguities. `http://javasrc.sourceforge.net`, 2002.

[32] S. Stanchfield and T. Parr. Parsers, part iv: A java cross-reference tool. `http://developer.java.sun.com/developer/technicalArticles/Parser/Series%Pt4`, 1997.

[33] Software visualisation group, department of computer science, university of canterbury. `http://www.cosc.canterbury.ac.nz/research/RG/svg`.

[34] M. Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, 1986.

[35] World wide web consortium. `http://www.w3c.org`.