Automatically generating consistent graphical user interfaces using a parser generator

Kristina P. Boysen and Gary T. Leavens

TR #04-07a
August 2004, revised November 2005

Keywords: automatic code generation, graphical user interface, consistency.
2001 CR Categories: H.5.2 [*Information Interfaces and Presentation*] User Interfaces — graphical user interfaces (GUI), interaction styles, standardization, theory and methods.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Automatically generating consistent graphical user interfaces using a parser generator

Kristina P. Boysen and Gary T. Leavens*
Iowa State University, Department of Computer Science,
226 Atanasoff, Ames, IA, 50011, USA

November 18, 2005

Abstract

Well-designed graphical user interfaces (GUIs) are needed to make programs easier to use. However, these programs are difficult and time-consuming to develop, especially when creating several GUIs for a related set of tools. Automatic generation of a GUI is one solution to this problem. This paper presents a technique to automatically generate GUIs using a parser generator and data files. This technique quickly creates multiple consistent GUIs for tools with similar options. In this way, programmers can quickly create several GUIs at once that look and act the same.

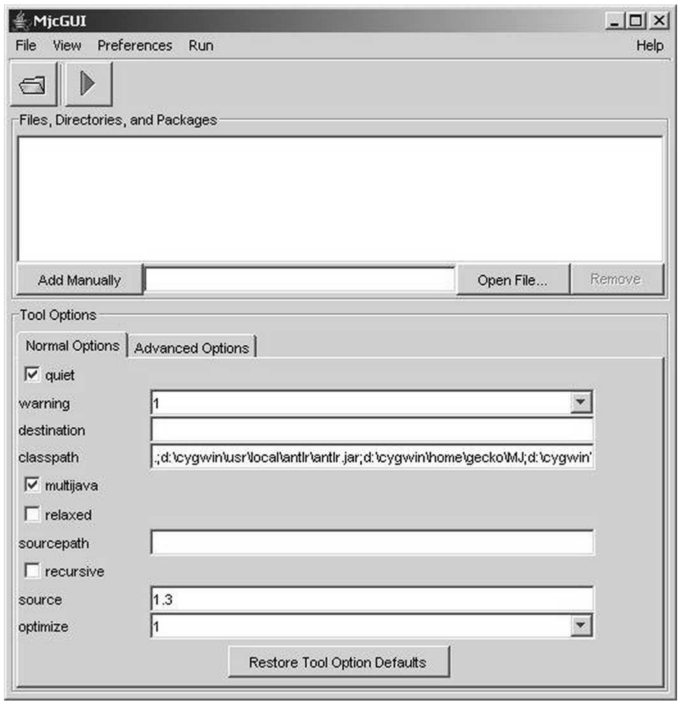
INTRODUCTION

An important part of a computer program is its user interface. Well-designed graphical user interfaces (GUIs) can make user experiences more effective and satisfying. To achieve these goals when a user has access to several related tools, consistency is especially important. To achieve consistency GUIs need to have consistent layout and behavior. However, making sure a GUI meets these requirements takes programmers' time, sometimes causing the GUI to be delayed or unavailable. In addition it must be easy to maintain and enhance these GUIs while maintaining consistency.

In this paper we present an approach to make sure that GUIs for a set of related programs all have the following attributes:

- GUIs are easy to maintain so that revisions can be made quickly and easily.
- GUIs are consistent so that users can find similar functions from one window to another.
- GUIs can be produced quickly so the client does not wait long for the finished product.

*The work of both authors was supported in part by the US National Science Foundation, under grant CCR-0113181.



[✓] quiet
warning: 1
destination:
classpath: ;d:\cygwin\usr\local\antlr\antlr.jar;d:\cygwin\home\gecko\MJ;d:\cygwin'
[✓] multijava
[ ] relaxed
sourcepath:
[ ] recursive
source: 1.3
optimize: 1

Figure 1: The GUI for the MJ compiler.

An example of the need for automatically generated GUIs arose in two projects: MultiJava (MJ) and the Java Modeling Language (JML). (The JML tools are built on top of the MJ compiler [1].) These projects already had complete command line option processing for each of their seven separate tools, but the project supervisors wanted to add GUIs to allow use of the tools without a command line. So, instead of using the command line command `mjc -q` to run the MJ compiler quietly, the user can go into the MJ compiler GUI and select the "quiet" option.

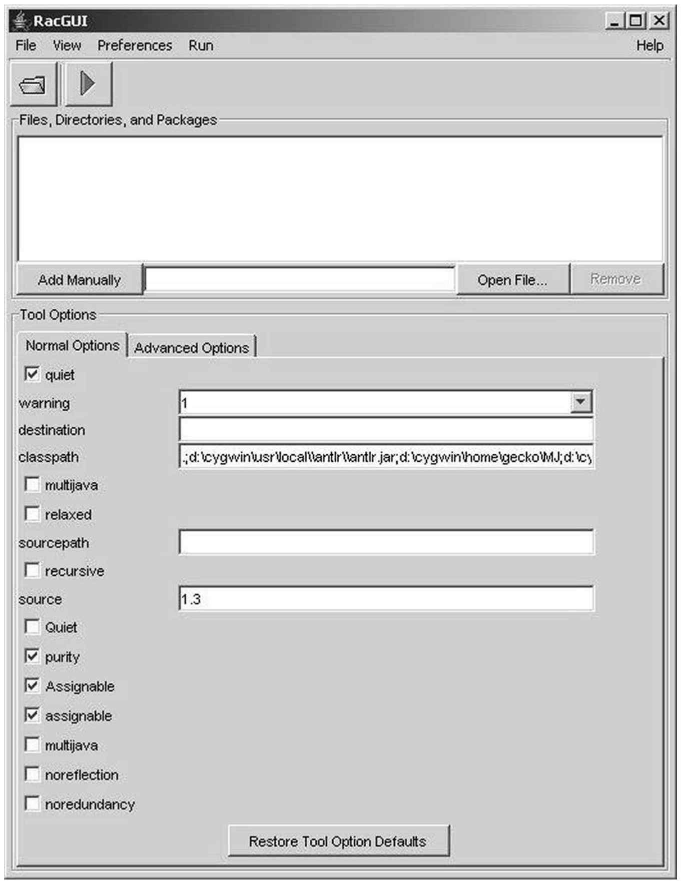
Since all of these tools were related to each other and had many similarities, the GUI had to have the same approximate layout and behavior. Each GUI needed a way for users to enter files, directories, and packages for the tool to process and to access to the different options of the tools. Figure 1 shows the final design of a typical GUI.

Compare figure 1 to figure 2 on the following page, which shows a similar GUI. Both GUIs have the same interface for entering files, directories, and packages, and even though some of the options are different, the layout of the options is the same for both GUIs.

Coding the GUI separately for each tool makes it more likely that all of the GUIs act differently from each other. This can confuse the user who expects the behavior to be the same throughout all of the tools. Maintenance is also difficult and time-consuming when changes need to be made to the look or behavior of the GUIs. Each GUI must be modified separately, increasing the chances for coding errors.

Our approach to solving this problem was as follows. First, one GUI was manually coded to provide the overall layout and behavior desired. Next this GUI was factored

[1] See www.multijava.org and www.jmlspecs.org for details about MJ and JML.



[✓] quiet
warning: 1
destination:
classpath: .;d:\cygwin\usr\local\\antlr\\antlr.jar;d:\cygwin\home\gecko\MJ;d:\cy
[ ] multijava
[ ] relaxed
sourcepath:
[ ] recursive
source: 1.3
[ ] Quiet
[✓] purity
[✓] Assignable
[✓] assignable
[ ] multijava
[ ] noreflection
[ ] noredundancy

Figure 2: The GUI for the JML Runtime Assertion Checking Compiler.

out into an (eventually shared) abstract superclass, while all of the fields that contained tool-specific information were factored out into GUI subclasses. Each subclass supplied tool-specific information by overriding methods that the abstract superclass relied on. This allowed each tool to define a different GUI with a consistent interface. These subclasses were then automatically generated. The current automatic generation process relies on a programmer-defined text file that is used by a parser. (The parser is itself automatically generated using a grammar and a parser generator.)

Automatic generation of GUIs could also be accomplished in other ways. For example, instead of using subclassing to customize a general superclass, one could use a factory class to create the different GUIs based on parameters passed into the factory's constructor. However, to solve the consistency problem using such a factory, one would still have to automatically generate code to call the factory's constructors. Thus, using such a factory would have similar benefits to using subclassing, only differing in the implementation details.

This paper covers the technique of automatic generation of this and similar GUIs using the MJ and JML GUIs as examples. The next few sections presents the different parts that make up the automatically generated GUIs, the results of the project, and an overview of the related work for this project. An appendix is included at the end of the paper that explains how a parser generator works, using ANTLR (ANother Tool for Language Recognition), the parser generator chosen for this project.

AN AUTOMATICALLY GENERATED GUI FOR MJ AND JML

From the programmer's perspective, an automatically generated GUI is created in two steps:

1. **Collecting commonalities** between all of the GUIs into an abstract superclass, which holds all of the GUI components. This superclass defines the overall layout and behavior of each GUI.

2. **Extracting differences** between all the GUIs, which are placed in special data files. These data files are used to automatically generate concrete subclasses that are instantiate for each separate GUI.

The next few sections will describe these steps in detail using our running example.

Collecting Commonalities Between the GUIs

In our example, two main commonalities exist between all of the desired GUIs: the GUI construction code and the tool options code.

GUI Superclasses

The GUI superclass contains the general layout that the GUI follows and defines the helper classes that the GUI will use. Since this abstract superclass has subclasses that inherit from it, the superclass defines abstract methods that specify the methods that each concrete subclass must implement. Abstract methods allow the superclass to call methods in the

```
protected GUI(String name, Options options,
boolean commandLine) {
  super(name + "GUI");

  setJMenuBar(createMenus());
  /* other details omitted */
}
```

Figure 3: GUI constructor example, with a call to setJMenuBar

subclasses without knowing how they are implemented. These ”hook” methods allow the subclasses to implicitly parameterize or specialize the abstract superclass. (This is an instance of the ”Template Method” design pattern [4].)

For example, our GUI has a link to the specific project's website. Since the name and location of that website changes between tools, the GUI needs to call methods in the subclasses that would know the name and location of the website. To see this in more detail, consider first figure 3, which shows how the constructor for the superclass, named GUI, is built. (For simplicity, all the GUI creation methods are omitted except for a simple menu creation tag.)

The superclass GUI code builds a simple menu bar for the GUI. The help menu, which contains the website's URL, is created on this menu bar in figure 4 on the following page.

Notice that there are two methods that are not yet defined: `getWebpageName()` and `getWebpageLocation()`. These are abstract methods that are only declared in the GUI superclass. Figure 5 on the next page shows the declarations of these abstract methods.

Tool Options Representation

To flesh out our example, we now discuss creation of the options system for the MJ and JML GUIs. We begin with a small amount of necessary background on the options available for these tools.

There are three types of command line options used in both MJ and JML tools:

- **Boolean options** expect a true or false value to turn that option on or off.
- **String options** expect a string to set that option to an open-ended user-defined value.
- **String List options** expect a list of strings to let the user choose between those predetermined values.

In order for the options to be selectable in a GUI, each option must be presented to the user. We chose to do the presentation using classes in Java's standard Swing framework:

- Boolean options are represented by a `JCheckBox`.
- String options are represented by a `JTextField`.

```
private JMenuBar createMenus() {
  JMenuBar menuBar = new JMenuBar();
  JMenu helpMenu = new JMenu("Help");

  JMenuItem aboutItem = new JMenuItem(getWebpageName());
  aboutItem.addActionListener(
    new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        aboutWindow = new GUIAboutWindow(
          getWebpageLocation());
      }
    }
  );
  helpMenu.add(aboutItem);

  menuBar.add(helpMenu);
  return helpMenu;
}
```

Figure 4: Help menu construction, with calls to abstract methods getWebpageName and getWebpageLocation.

```
protected abstract String getWebpageName();
protected abstract String getWebpageLocation();
```

Figure 5: Abstract method declaration

- String List options are represented by a `JComboBox`.

Recall that the goal of the GUI effort is to provide a way for users to access the different command line options in the tools and make them easily selectable. Each GUI is technically a (Swing) graphical "container" with all of these options represented by objects of the classes named above. The GUIs use access methods to communicate the user's changes and to manipulate the rest of the program's (command line) options in that program's internal representation. Thus, in terms of the Model-View-Controller design pattern [4], each GUI is both a "view" class and a "controller" class, with the rest of the program playing the role of the "model".

Extracting and Representing the Differences Among the GUIs

In applying the second part of our technique to our example, we identify seven major differences between each tool in the two projects:

- **Prefix**: The three letter identifier that differentiates each tool
- **Package**: The folder that the tool (and consequently the GUI) is in
- **Imports**: Any extra classes the subclass needs to access
- **Readable Files**: A string containing the description of what kind of files can be read into this tool
- **Webpage Name**: The name of the project's webpage
- **Webpage Location**: The URL of the project's webpage
- **Accepted Suffixes**: A list of strings of filename suffixes that can be accepted by this tool

All of the subclass code for each GUI, including the bodies of methods that override the declared abstract methods from the GUI superclass, is automatically created from a file that specifies values for these seven differences. Generation of the subclass code is done using ANTLR and a grammar to define the parsers to process an input file (see the appendix for more information). In this file, the seven differences listed above are represented in the format exemplified in figure 6 on the following page.

The parser, created by ANTLR, generates a Java class (in source form) containing the GUI subclass. This class contains the GUI's `main` method and overrides all the abstract methods required. The `main` method calls methods of the GUI superclass, which in turn call back down to the subclass for specialized behavior.

After this initial work was done, input files are built for all of the tools that needed GUIs. These input files contain the main GUI-related differences in the final code.

Connecting the GUIs to the tools is accomplished by changing each tool's `Main` class [2]. Since these `Main` classes already subclass the `Main` class used by the MJ compiler, this process involves writing only a few methods in each `Main` class.

[2]The Main class for each tool runs the command line interface for that tool.

```
prefix            Jml
package           org.jmlspecs.checker
imports           org.multijava.util.gui.GUIUtil
readableFiles     "JML Readable Files"
webpageName       "JML Web Page"
webpageLocation   "http://www.jmlspecs.org"
acceptedSuffixes  "jml", "refines-jml", "jml- refined",
                  "spec", "refines-spec", "spec-refined",
                  "java", "refines-java", "java-refined"
```

Figure 6: Input file format example

ADVANTAGES AND LIMITATIONS

Our example GUIs for the MJ and JML tools show that automatically generated GUIs solve the problems we describe in the introduction:

- **Automatically generated GUIs are easier to maintain** because there are only two pieces of common code. In our case, maintenance typically only requires changes in the GUI superclass. Every change made to this superclass is reflected in each tool's GUI, so the number of files that need to be modified is lowered. This also cuts down on errors in the source code, since all common code is collected into one superclass and not copied into several subclasses. In addition, there is only one place where the subclass code is generated, so changes to subclasses, such as addition of new kinds of differences, is also easy.
- **Automatically generated GUIs are more consistent** because they are generated from a common structure and a minimal set of differences. In our example, the common structure is defined in one common superclass. Since this superclass defines both the layout and behavior of each of its subclasses, each GUI appears and behaves the same. This helps new users to find and use GUI functions quickly since all of the GUIs are generally the same.
- **Automatically generated GUIs are produced quickly** because once the common code and the generation code is written, only simple input files that describe the differences need to be written for each GUI. The parser for these input files creates the needed subclasses automatically, saving time for maintenance and testing.

Even though it has major advantages over separate creation of individual GUIs for a set of related tools, there are limitations to our technique. The main limitations are that our technique is only applicable if:

- There are a small number of kinds of differences among corresponding windows from each of the desired GUIs, and
- The differences between windows are easily encoded into data files.

Therefore, our approach would not be the best solution if either the desired GUIs have a large number of kinds of differences among corresponding windows or if these differences are not easily in coded into various data files. When the problem does not conform to our technique's limitations, the number of commonalities in the superclass would be so small that most of the errors would come up in the highly specialized subclasses, making the program hard to maintain. We are able to accommodate a large number of differences in the MJ and JML command line options since all options are processed in a similar manner and there are only a limited number of different kinds of options.

When the problem falls outside the limitations of our technique, then a better alternative to our approach may be to design a new GUI framework. A framework shares common code in several superclasses, and is designed to allow easy extension by subclassing. When the differences between GUI windows are large or when there are many differences that cannot be reduced to data input files, then writing the individual subclasses by hand is a sensible alternative. In such cases it may be clearer to separate the differences into subclasses written by hand. In some sense, the commonalities are collected into the framework, which is a more elaborate version of our common superclass, and the differences are extracted into the various subclasses, which are more elaborate and complex versions of the little language that parses the data files in our technique. Looked at in this way, our technique can be seen as a special case of the more general technique of writing a framework and instantiating it to create the GUIs. However, the advantage of our technique is that it is considerably easier to apply and also enforces greater consistency between the GUIs.

PROBLEMS ENCOUNTERED

We encountered a few small problems with refactoring our prototype GUI (for the MJ compiler) in order to make it into a superclass that collected the commonalities among all the GUIs for the six other MJ and JML tools. One of the problems was how the "Accepted Suffixes" option was handled. At first, the GUI was tested only with the MJ compiler. Since the MJ compiler can only accept Java source files with suffix `.java`, the GUI could not run the other tools in JML properly, since they add more suffixes such as `.java-refines` or `.jml`. These were already presented in a list of strings in the input file, but they were not properly processed between the input file and the ANTLR grammar, thus prompting a change in the ANTLR grammar. However, this example shows that previously unrecognized differences can be accommodated in our approach, provided they are small and reducible to data files.

RELATED WORK

Research into automatically generated GUIs is separated into several different categories. Some focus on creating a GUI from a specific data set, such as from a database. Iizawa and Shirota [6] note that these GUIs suffer from simplistic and inadequate layouts, in addition to having GUI production algorithms that are complicated and difficult to implement. Some solutions to this problem, such as Kasuga Script [6] and DIGBE [9], involve generating the GUI based on user preferences. The idea of automatically generating GUIs from sets of data leads to GUIs that are more or less consistent, and so if these systems were tailored to options processing, we could have used them to solve our problem. However, the authors do not mention consistency or relate it to their automatic generation technique. This recognition is one of the contributions of the present paper.

Other solutions additionally take into consideration the type of device the GUI is currently running on. The SUPPLE system [3] uses a functional interface language that describes the GUI's functions, as well as a user and device model, to create multiple interfaces for the same application. In the given example, the authors are able to modify an application controlling auditorium lighting for a normal computer screen, a touch screen, and a cell phone screen. In addition, the interface slowly changes according to the user's usage patterns so that the best interface is displayed each time the user accesses the program. For example, the authors were able to use a login name text field in the first usage of an FTP client, but when several people logged into the FTP client successively, the text field changed to a drop down list of previous users. Mostly these changes are small, but SUPPLE does not put bounds on the consistency between interfaces because of the way the separate interfaces are formed by interactions with the users. Thus, SUPPLE does not meet our needs because it addresses making adaptable GUIs for different devices, not consistency between multiple similar programs on the same device.

While the major focus of the SUPPLE research is on the adaptability of user interfaces between separate devices, it also defines a modular representation of GUI data. Several systems use this concept, including the COUSIN system [5], the model-based interface designer HUMANOID [11], the ART toolkit [8], the domain model interface generator Mecano [10], and a tool for generating plug-in GUIs from formal specifications in Abstract Syntax Notation One (ASN.1) form [2]. However, none of these guarantee consistency between GUIs for related applications, as does the technique we describe in this paper. We could use any one of these systems as a back-end to our automatic generation technique, but using such a system would not, in itself, guarantee consistency among all the GUIs for all of the MJ and JML tools we worked with.

Other projects emphasize the consistency problem. ITS [12] is an interface designer splitting design concerns into four parts: the action layer, the dialog layer, the style rule layer, and the style program layer. The layers relevant to our problem are the last two, which specify the interaction of the data and the layout of the widgets representing the data, respectively. This helps ensure consistency inside a particular GUI between different subparts of that GUI. For example, users can specify conditions on a dialog that cascade down to children of that dialog. The authors state that these conditions are consistent because they are "represented and interact with the user similarly wherever found in an interface" [12]. Similarly, Zhou and Feiner include "consistent design within and across displays" in their definition of *coherent visual discourse* [13]. They attempt to achieve such consistency by monitoring the user's tasks so that one task performed several times will always be presented to the user in the same way. When the GUI is called on to perform a new task, their system determines whether the new task is similar to any other task it has performed previously. If so, then the system instructs the GUI to present the new task in a similar format as previous similar tasks. This technique ensures some consistency between tasks within a particular GUI. It is an open question whether this process would ensure consistency between different GUIs, which is the problem we address.

FUTURE WORK

One area of future work is to try to validate our approach using a different case study. One way to do this would be to apply our technique to several Unix commands. The idea would be to handle their command-line options visually in a consistent manner. This would test the generality of our technique and how easily it could be applied to other families of programs with structured options.

A related area of future work is to test whether a consistent family of GUIs could help users be more productive. The proposed case study for Unix commands could be a good way to test this hypothesis.

Another interesting area of future work is to study the differences between the two different implementation techniques that we describe for our approach. Recall that our implementation uses subclassing to extend a framework to specialize each GUI. The alternative implementation technique is to have a set of generators and to use parameters to specialize each GUI. The point of the experiment would be to try to get an understanding of which implementation technique is easier and in what circumstances.

CONCLUSIONS

The technique of automatically generating a set of related GUIs met the consistency requirements for the tools in MJ and JML. We were able to apply this technique, since these tools have a small number of kinds of differences. This case study provides evidence that the use of automatically generated GUIs is applicable in such situations. In our example, all of the GUIs have the same kinds of graphical objects in the same places on the GUI window, which presents a consistent interface to users.

Our experience with the approach described in this paper has been very positive. We were able to quickly create GUIs for several tools, and the GUIs created are consistent and easy to maintain. We believe that the same approach would work for other families of related GUIs.

ACKNOWLEDGEMENTS

Thanks to Curtis Clifton, the supervisor of the MJ project, for several discussions that led to this approach. Thanks to David Cok, who provided valuable suggestions on the design and behavior of the GUIs. Finally, thanks for all the GUI design suggestions from the people on the Sourceforge MJ and JML mailing lists.

References

[1] ANTLR Parser Generator.
http://www.antlr.org [28 October 2005].

[2] Ceterberg P. Automatic GUI generation - generating Java source code from formal descriptions.
http://www.percederberg.net/home/essays/master/master.pdf [28 October 2005].

[3] Gajos K, Weld D. Automated user interface generation: SUPPLE automatically generating user interfaces. In *Proceedings on the 9th International Conference on Intellegent User Interfaces*, Funchal, Madeira, Portugal, January 2004, 93–100.

[4] Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley: Reading, Mass., USA, 1995;

[5] Hayes PJ, Lerner RA, Szekely PA. The COUSIN user interface project. *ACM SIGOA Newsletter* 1983; **4**(2):3–7.

[6] Iizawa A, Shirota Y. Automatic GUI generation from database schema information. *Systems and Computers in Japan* 1997; **28**(5):1-10.

[7] jGuru
http://www.jguru.com/faq/view.jsp?EID=78. [27 October 2005]

[8] Olsen DR Jr., Holladay W. Automatic generation of interactively consistent search dialogs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Celebrating Interdependence*, Boston, Massachusetts, USA, April 1994, 218–224.

[9] Penner RR, Steinmetz ES. Dynamic user interface adaptation based on operator role and task models. In *2000 IEEE International Conference on Systems, Man, and Cybernetics*, Nashville, Tennessee, USA, October 2000, 1105–1110.

[10] Puerta A, Eriksson H, Gennari J, Musen M. Model-based automated generation of user interfaces. In *Proceedings of the National Conference on Artificial Intelligence*, Menlo Park, California, USA, August 1994, 471–477.

[11] Szekely P, Luo P, Neches R. Beyond interface builders: Model-based interface tools. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Amsterdam, The Netherlands, April 1993, 383–390.

[12] Wiecha C et. al. ITS: A tool for rapidly developing interactive applications. *ACM Transactions on Information Systems* 1990; **8**(3):204–236.

[13] Zhou M, Feiner S. Top-down hierarchical planning of coherent visual discourse. In *Proceedings of the 1997 International Conference on Intelligent User Interfaces*, Orlando, Florida, United States, January 1997, 129–136.

APPENDIX A. BACKGROUND ON ANTLR

ANTLR [1] is a parser generator which uses a grammar in Extended Bakus-Naur Form (EBNF) to create lexers and parsers for translating text files into Java, C++, or C# code. ANTLR needs two files to run: a grammar file and an input file. The grammar file defines the format of the input file in EBNF format. ANTLR produces a lexer and parser from the grammar file. A lexer breaks up characters from the input file into units called tokens, and a parser works on these tokens, producing the desired output data. This process creates

```
"prefix"            prefix            = aIdentifier[]
"package"           packageName       = aName[]
"imports"           imports           = aNameList[]
"readableFiles"     readableFiles     = aString[]
"webpageName"       webpageName       = aString[]
"webpageLocation"   webpageLocation   = aString[]
"acceptedSuffixes"  acceptedSuffixes  = aStringList[]
```

Figure 7: ENBF format for part of the automatically generated GUI grammar.

```
aString []
returns [String str]
        :  t : STRING
                { str = t.getText(); };
```

Figure 8: Definition of aString[].

the subclasses of the automatically generated GUI. The next sections use an example to explain how the lexer and parser are created and how these create a Java file that can be used to define the GUI.

THE ANTLR LEXER AND PARSER

From an EBNF grammar, ANTLR creates two classes: a lexer and a parser. Both of the specifications for these come from the grammar, which is called "Guigen" in this project. The lexer simply needs definitions of what to expect in the input file, like the format for a string, in order to create a series of tokens for the parser. The parser needs a definition of how these tokens are organized in the input file so it can create a datatype out of them. An example is the Guigen grammar shown in figure 7.

Notice how the field `webpageLocation` is defined in the second to last line of figure 7. The leftmost column is the keyword that the parser is looking for in the input file, the middle column is a variable name used only by the parser, and the rightmost column is the type of the field.

Types are partially defined by the lexer; for `webpageLocation`, the lexer defines what should be defined as a `STRING` in the input file. The parser then defines the method `aString[]` based on the definition of `STRING`. The "`[]`" following `aString` represents a call to that method with no arguments. Figure 8 shows the grammar of `aString[]`. The method returns `str`, which has the value of the text from `STRING`.

Now the only thing needed to do is to define the constructor of the datatype, as shown in figure 9 on the following page. The lexer and parser created by ANTLR return an instance of this datatype, but a `DefinitionFile` must be hand built to specify methods that need to be included in the new datatype.

```
self = new DefinitionFile(sourceFile , packageName, imports,
                          parent , prefix , readableFile , webpageName,
                          webpageLocation , acceptedSuffixes);
```

Figure 9: The datatype constructor.

```
public DefinitionFile(String sourceFile ,
                      String packageName,
                      String imports,
                      String prefix ,
                      String parent,
                      String readableFiles ,
                      String webpageName,
                      String webpageLocation ,
                      ArrayList acceptedSuffixes)
```

Figure 10: Constructor for the DefinitionFile that defines the datatype.

CREATING THE DATATYPE

Now that the Guigen grammar is specified, the `DefinitionFile` must be coded by hand. The format found in figure 6 on page 8 is used to specify the constructor in 10.

`DefinitionFile` should know how to instantiate the lexer and parser to read the input file. A standalone program, called `Main`, now can tell `DefinitionFile` to process an input file without `Main` knowing about the lexer or parser. `DefinitionFile` should also know how to create a Java file using the variables parsed from the input. For example, `webpageLocation` needs an access method so other classes can access the URL. The access method is printed out to the new Java file using `out.println()` statements, as shown in figure 11.

RUNNING THE PROGRAM

Now that ANTLR has created both the lexer and parser and the user has created `DefinitionFile`, a Java `Main` program can use them. First, the `Main` program creates a `DefinitionFile` type by telling it to process the input file. The input file must be in a format that the lexer

```
out.println();
out.println("protected String getWebpageLocation() {");
out.println("return \"" + webpageLocation + "\";");
out.println("}");
out.println();
```

Figure 11: Print statements used to create the webpageLocation method for the output file.

and parser expect (defined by figure 7 on page 13). In the example of this format in figure 6 on page 8, notice that `webpageLocation` is a `String` as defined by the grammar file.

After this file has been fully translated into a Java file, the Java file can be used to access parts of the `DefinitionFile` type, such as `webpageLocation`. This is a fast and easy way to create and modify code, since data changes only have to be made in the input files, and the grammar defining the input files is easy to change correctly.