# Visual Classgen

visually aided design of abstract data types

Version 1.0

# User's manual

S. Winter

F. Deißenböck

supervision:
Alfons Brandl

# Contents

**1 Introduction** **3**
1.1 What is vcg? . . . 3
1.2 About this document . . . 3
1.3 From classgen . . . 3
1.4 . . . to vcg. . . . 4
1.5 The 1st example . . . 4

**2 The specification language** **6**
2.1 Basic data types . . . 6
2.1.1 Records . . . 6
2.1.2 Variants . . . 8
2.1.3 Lists . . . 9
2.2 Enumerations, attributes and methods . . . 10
2.2.1 Enumerations . . . 10
2.2.2 Attributes . . . 10
2.2.3 Methods . . . 11
2.3 Formal specification syntax . . . 12
2.4 Combining data types . . . 12
2.5 Visitor design pattern . . . 13

**3 The vcg user interface** **17**
3.1 Pull down menus and tool bar . . . 18
3.1.1 Menu bar . . . 18
3.1.2 Tool bar . . . 19
3.2 Specification editor . . . 19
3.3 Output window . . . 19
3.3.1 Diagram pane . . . 19
3.3.2 Class tree . . . 20
3.3.3 Messages . . . 20
3.4 Online help . . . 20

**4 Examples** **22**
4.1 vcg syntax . . . 22
4.2 Spreadsheet . . . 24
4.3 Compiler . . . 26

**5 Known problems** **28**

**References** **28**

# 1 Introduction

## 1.1 What is vcg?

By definition vcg is a tool for the "visually aided design of abstract data types". Ok, but what does that mean?

vcg gives the possibility to generate JAVA classes for data types from a description written in a simple specification language. On the way from the textual description to the JAVA classes vcg supports the developer with a graphical, tree-like representation of the (recursive) data types. vcg uses grammars as specification language, so vcg is well-suited for compiler design, e.g. for the definition of abstract syntax trees.

vcg has a rich list of features including

*Editor* Full-featured specification editor with syntax highlighting.

*Visitor pattern* vcg has built-in support for the "visitor pattern"

*Instant feedback* vcg is multi-threaded and uses a second thread to auto-update the output window. You just have to stop typing for short time and vcg issues a re-parse of the specification.

*Tree-like representation* For a clearer representation vcg uses a tree-like structure to display the data types. Due to vcg's layout algorithm you can handle even large specification without loosing track of your data types.

*Direct manipulation* The data type representation supports direct manipulation, so you can modify the graph if you're are not fully satisfied with the auto-layout.

*Diagram export* vcg supports a JPEG-export of the current specification. You can use this for documentation or teaching.

*Online help* vcg has a built-in HTML-based online-help.

*Error messages* Error messages are visualized in a messages pane. The messages pane gives a description of the error and holds information on line of occurrence.

vcg combines the speed and flexibility of textual input with the ease of use of a visual development tool.

## 1.2 About this document

This document is meant as a users' manual. It covers all the feature of vcg and gives a detailed description of the specification language. The following table gives an overview of this document.

| | |
|---|---|
| **Section 1** | introduces vcg and classgen. Gives a small example. |
| **Section 2** | detailed description of vcg specification language and the "visitor pattern". |
| **Section 3** | detailed description of vcg user interface. |
| **Section 4** | shows three rather big examples with specification and representation. |
| **Section 5** | a list of problems vcg still has. |

If you are interested in details of the implementation of vcg please refer to [2].

## 1.3 From classgen ...

For a better understanding of the concepts of vcg we have to introduce the command line tool classgen of [1], which is ancestor and integral part of vcg. classgen does exactly the same as vcg but without a graphical representation of the data types. It allows to generate JAVA classes from a input file with different productions describing the data types. Therefore we have a two-step design process (fig. 1).

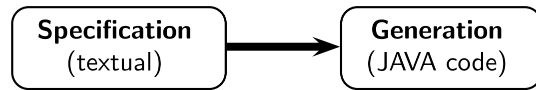



Figure 1: The two-step design process of classgen

## 1.4 . . . to vcg.

vcg adds a third level to this design process, the visualization (fig. 2).

Figure 2: The three-step design process of vcg

Now you can guess, what vcg stands for: visual classgen. It displays the different productions which describe the data type in a tree-like graph. As you will see in our 1st example the additional step of visualization chiefly improves design efficiency and ease of use.

Working with vcg you will see the advantages of the visualization step. Before generating the JAVA classes you have the chance to detect syntactical and logical problems in your design. Syntactical errors aren't a problem. The generator will simply produce something like “syntax error”, that's nothing new. classgen would have done this, too. Logical errors a more sophisticated. The syntax is correct but you used the specification language in a wrong way. With classgen you would have detected logical problems only by examining the generated code. Due to the graphical representation of vcg you will recognize logical error before you even generate code.

## 1.5 The 1st example

Before taking a closer look on the specification language and the features of vcg, let's start with a small example.

Consider the following situation. You're about to program a small tool to store information about your CD collection. You'll definitely need a data type, a JAVA class which stores information about a single CD. We will call this data type `CD`. Which information do we have to store about a CD? For our small example we will save the following values:

1. A CD must have an artists, which is stored in a string.
2. It must have on of the genre classifiers “Rock”, “HipHop”, “Punk” or “misc”.
3. It must have a list of tracks, which itself have titles stored in strings.

Here we have to introduce the three basic data type definitions vcg knows. We'll do this step by step. You may have slight problems of understanding at this point but don't worry things will become clear in the next section. We will concentrate on the visual representation, the specification language is topic of the next section.

Surely `CD` is the head of the structure, it has to store three different types of information, a string for the artist, a list of tracks and the genre. `CD` holds different information, we call it a *record*. Its graphical representation looks like (fig. 3). As you can see each element of the record has a *selector*. The selectors

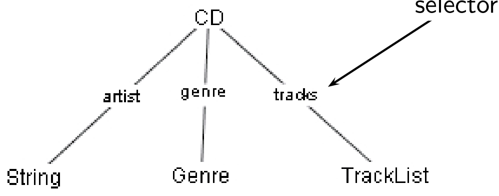



Figure 3: a simple *record*

will be the names of the variables in the generated JAVA code.

For storing the genre we build a base class `Genre`, one of the elements of `CD`, and four sub classes for the genres. We call such data types *variants* or *alternatives* and draw them like (fig. 4).

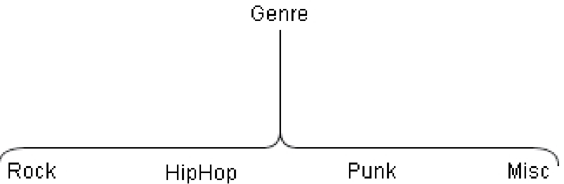



Figure 4: a *variant*

The third data type has to store a collection of items of the same type. In our example `TrackList` stores `Track`s. We represent this data type with a double line (fig. 5) and call it a *list*.

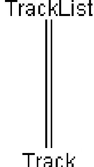



Figure 5: a *list*

For a clearer representation we can put all these diagrams together (fig. 6). Additionally we have another record (with just one element) for the tracks. The diagram suggest that we are displaying a tree. But this is not correct, each of the sub-data-types is generated by one production in the input file. We only put them together to clarify the representation.

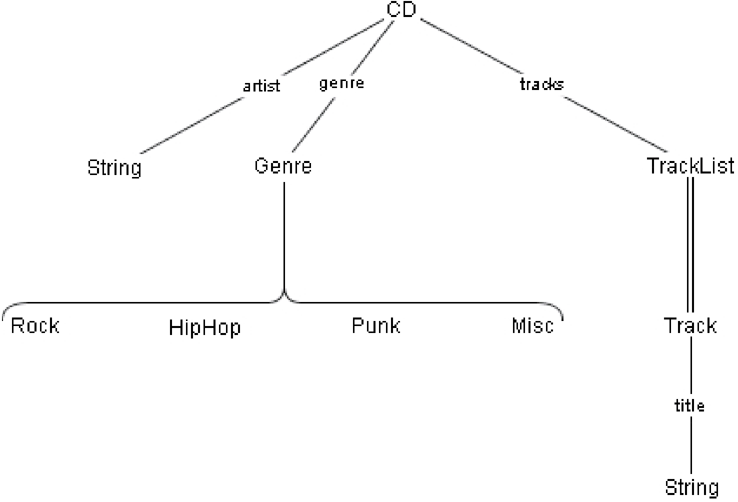



Figure 6: the complete diagram for `CD`

# 2 The specification language

This chapter deals with the specification language of vcg and classgen. They both use exactly the same language which uses grammars to specify data types.

Since vcg uses grammars to describe the data types the specification is (mainly) a list of productions. All productions have the form of

`left-hand-side ::= right-hand-side`.

The different data types are introduced in an informal way, for a formal definition of the specification syntax see (2.3).

## 2.1 Basic data types

The description extends the 1st example (see 1.5) and discusses the basic data types, their visualization and the generated code.

### 2.1.1 Records

As said in the 1st example we call a data type which contains different elements a *record*. We go back to our CD database again. We want the CD to store two values, the artist as a string and the length as an object of the JAVA class `Time`. Therefore we have a record with two elements and the specification and visualization would look like that.

```
package cdDataBase;

CD::= String Time
```

With the `package cdDataBase` command we make our class member of the `cdDataBase` package. The `package` command is not mandatory. You can generate classes which aren't members of a package.

You can see, the specification of a record is very simple. We just specify that the class `CD` should have two members, one of object `String` and one of `Time`. We take a look on part of the code vcg generates for the `CD` class.

```
package cdDataBase;

public class CD {

  private String string;
  private Time time;

  public CD (String string, Time time) {
    this.string = string;
    this.time = time;
  }

  public void setString(String string) {
    this.string = string;
  }

  ...

  public Time getTime() {
    return time;
  }

  public String toString() {
```

```
    ...
  }
}
```

Seems like everything is like we wanted it to be. The class is called `CD`, it is in the package `cdDataBase`, it has two instance variables (`String, Time`), the corresponding get/set-methods, a constructor and a `toString()`-method. There is only one problem, the `String` element which should hold the artist hasn't got a very intuitive variable name. Its simply called `string`. Here we can use the selectors, already introduced in the $1^{st}$ example. We can specify the data type like that (we omit the `package` command from now on).

```
CD::= string:artist
      time:length
```

We now explicitly specified the selectors (which will be the names of the variables) and called it `artist` and `length`. As you saw in the last example you needn't have to do it. If you don't specify the selector, `vcg` generates one of its own. As we hoped we get the following code.

```
public class CD {

  private String artist;
  private Time length;

  public CD (String artist, Time length) {
    this.artist = artist;
    this.length = length;
  }

  public String getArtist() {
   ...
}
```

Taking a closer look you will see that we made another change to the specification. We changed the writing of the name of the elements to lower case. `vcg` tries to follow the JAVA naming conventions and changes the first letter of class names to upper case.

This leads us to a slightly different notation if you like to use JAVA primitives like `int` or `boolean` which begin with a lower case letter. If you surround a specifier with double quotes `vcg` doesn't do any changes to the case. Suppose we want to add another element to our `CD` to store a CD number. We like to use a `int` and the specification looks like this.

```
CD::= string:artist
      time:length
      "int":number
```

The beginning of the code goes like this

```
public class CD {

  private String artist;
  private Time length;
  private int number;

  ...
 }
```

### 2.1.2 Variants

In the 1st example we equipped our CD with a element Genre. This element was a *variant* which could be on of the genres "Rock", "HipHop", "Punk" or "misc". We display this data type with a horizontal brace. We use different examples since the CD is not well-suited for further discussion.

Our new example deals with structures used by layout algorithms (like the algorithms of TEX). Common layout algorithms use boxes to arrange the content on a page or screen. For our example we use three kinds of boxes, a horizontal box (HBox), a vertical box (VBox) and a empty box (EmptyBox). HBox and VBox can again contain boxes, EmptyBoxess cannot. We start with this basic example and extend it step by step to explain all the features of variants.

```
Box::= {HBox}
     | {VBox}
     | {EmptyBox}
```

Let's take a look at the code first. vcg generated 4 files Box.java, HBox.java, VBox.java and EmptyBox.java, where Box is the abstract base class for HBox, VBox and EmptyBox.

**Box.java**

```
public abstract class Box {

  public String toString() {
    return print("");
  }

  public abstract String
    print(String tab);
}
```

**VBox.java**

```
public class VBox extends Box {


  public VBox () {
  }

  public String
    print(String tab) {
    ...
  }
}
```

**HBox.java**

```
public class HBox extends Box {


  public HBox () {
  }

  public String
    print(String tab) {
    ...
  }
}
```

**EmptyBox.java**

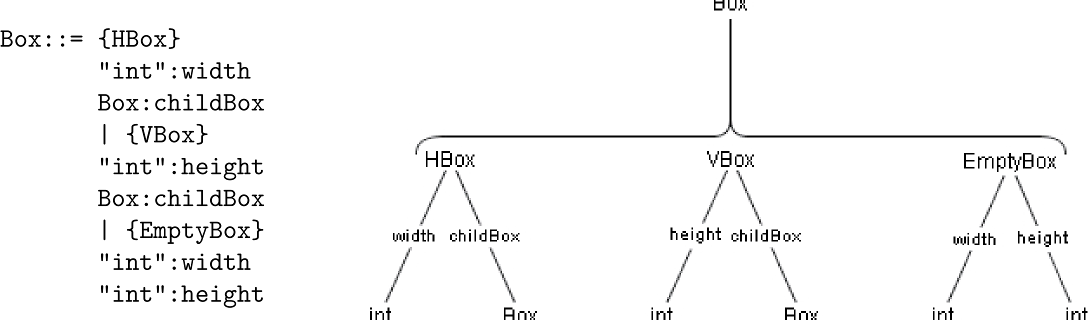
```
public class EmptyBox extends Box {


  public EmptyBox () {
  }

  public String
    print(String tab) {
    ...
  }
}
```

Since the boxes don't contain any information there rather useless. A HBox should have an int element to store its width. A VBox should have an int element to store its height and EmptyBox should store both values. Furthermore HBox and VBox can contain another Box. Implementing this is easy.

```
Box::= {HBox}
       "int":width
       Box:childBox
       | {VBox}
       "int":height
       Box:childBox
       | {EmptyBox}
       "int":width
       "int":height
```




The new HBox class has an int variable to store its width and contains another box the, childBox. Code for the other boxes looks similar. EmptyBox has two int variables and no childBox.

```
public class HBox extends Box {

  private int width;
  private Box childBox;

  public HBox (int width, Box childBox) {
    ...
  }

  public int getWidth() {
    return width;
  }

  ...
}
```

### 2.1.3 Lists

Back to our CD database example. A CD has more than one track, so we could not use records for storing the tracks, especially because we don't know the number of tracks. If we have an object which has references to many object of the same type we use *lists*.

The implementation of a list type with vcg is straight-forward. We use the ∗-symbol to create lists and display them with double lines.

```
TrackList::= Track*
```

This means TrackList can contain objects of the class Track. vcg implements sequences with java.util.vector. It generates on file TrackList.java which has a private Vector called items to store the Tracks. There are several convenience methods to access the items.

```
import java.util.Vector;
import java.util.Enumeration;

public class TrackList {

  private Vector items;

  public TrackList() { items = new Vector(); }

  public TrackList(Track anItem) {
    this();
    append(anItem);
```

```
  }

  public TrackList append(Track anItem) {
    if (anItem == null) return this;
    items.addElement(anItem);
    return this;
  }

  public Enumeration elements() {
    return items.elements();
  }

  ...

  public boolean isEmpty() { return items.isEmpty(); }

  ...
}
```

## 2.2 Enumerations, attributes and methods

Attributes, enumerations and methods are not visualized in the vcg class diagram. But attributes and enumerations are listed in the vcg class tree (3.3.2).

### 2.2.1 Enumerations

vcg offers another data type, enumerations. Enumerations are described best with an example. In our CD example we can use an enumeration as a different (better) way of storing the genre. We redefine the CD record as follows.

```
CD::= String:artist
      TrackList:tracks
      "int":genre
GenreConsts::= one of
               Rock, HipHop,
               Punk, Misc
```

We now use an `int` to store the genre, but what kind of data type is defined with the `one of` keyword? As you see the enumeration is not visualized in the diagram, but it is listed in the class tree (3.3.2). The second production does not produce a class, it produces an interface (figure on the right shows class tree representation):

```
public interface GenreConsts {

  public final static int Rock = 0;
  public final static int HipHop = 1;
  public final static int Punk = 2;
  public final static int Misc = 3;

}
```

You can use these `final static ints` to specify the genre of a `CD`

```
CD myCD = new CD("AC/DC", new TrackList(), GenreConsts.Rock);
```

### 2.2.2 Attributes

vcg offers another feature with is often used in compiler design and with attributed grammars. You can define attributes with name and type for every non terminal of the specification. In the following typical example you'll see how attributes work. Since attributes are only visualized in the class tree there's no diagram representation for this example.

```
attr DeklInfo deklInfo with Var;

Expr::={BinExpr} Expr:lhs
         "int":OP
         Expr:rhs
       | {Var}
Var::= String:ident
```

Var extends Expr
String:ident
DeklInfo:deklInfo
Expr
BinExpr extends Expr
Expr:lhs
int:OP
Expr:rhs

vcg generates the following code for `Var`. You'll notice difference between a attribute and an element of a record. The attribute is not a parameter of the constructor.

```
public class Var extends Expr {

  private String ident;
  private DeklInfo deklInfo;

  public Var (String ident) {
    this.ident = ident;
  }

  public String getIdent() {
    return ident;
  }

  ...

  public DeklInfo getDeklInfo() {
    return deklInfo;
  }

  ...
}
```

### 2.2.3 Methods

You can equip your classes with custom methods. This could be done with the `methods of` key word. This is also useful if only want to setup a method frame without body as a reminder. To equip our `CD` record with two methods for sorting the tracks (not very useful), you'll use the following code. As said before methods are not visualized by vcg. Generated JAVA code is on the right side.

**Specification**

```
CD::= String:artist
      TrackList:tracks
      Genre:genre
...

methods of CD{
  public void
    sortByName(){

    //quicksort
  }
  public void
    sortByLength(){

    //bubblesort
  }
}
```

**JAVA code**

```
public class CD {

  private String artist;
  private TrackList tracks;
  private Genre genre;

  ...

  public void sortByName(){
    //quicksort
  }

  public void sortByLength(){
   //bubblesort
  }
  ...
}
```

## 2.3 Formal specification syntax

We introduced the vcg specification language in a rather informal way. Table 1 gives a formal description of the specification syntax .

```
specification ::= [package] [attribute]* [production]* [method]*

package       ::= "package" Ident ["." Ident]* ";"

production    ::= Ident "::=" alternatives [attribute]*
                | Ident "::=" "one" "of" Ident ["," Ident]*

alternatives  ::= [ alternative ["|" alternative]* ]

alternative   ::= "{" Ident "}" [item]*

item          ::= Ident [":" Ident]
                | Ident "*" [":" Ident]

attribute     ::= ("inherited"|"synthesized"|"attribute")
                    Ident Ident ["," Ident]* ["with" [Ident]*] ";"

method        ::= "methods" "of" Ident "{" [line]* "}"

line          ::= [legal Character]*
```

Table 1: vcg specification syntax

## 2.4 Combining data types

Taking a closer look at the specification syntax (2.3) shows that there are more possibilities to combine the data types. We can define the `CD` data type with only one production.

```
CD::= String:artist
      Track*:tracks
```

CD
artist
tracks
String
TrackList
Track

vcg automatically generates a class `TrackList`. Additionally we can declare attributes directly in a variant production (class tree representation on the right).

```
Box::= {HBox}
       | {VBox}
       | {EmptyBox}
       attr boolean valid;
```

Variants have another feature. If you omit the braces around a sub class vcg automatically generates a record type sub class.

```
Box::= TBox:sel1
       UBox:sel2
       | {VBox}
       | {EmptyBox}
```

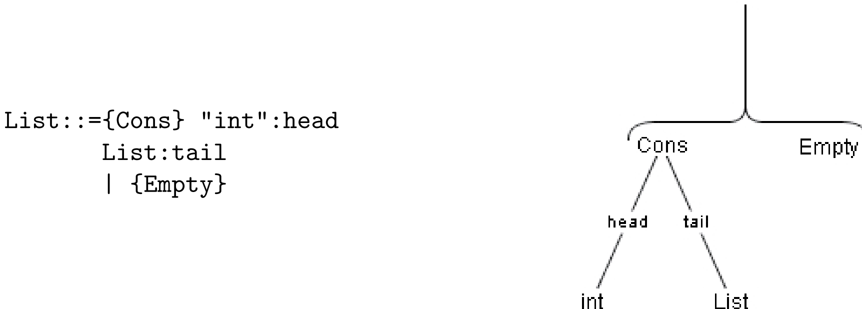
Box
Box0 EmptyBox VBox
sel1 sel2
TBox UBox

## 2.5 Visitor design pattern

After generation the JAVA classes can store their values but the don't have any functionality. To add functionality we can simply add methods and variables but two problems arise.

1. If we extend the vcg generated classes manually we must be absolutely sure that there will be no changes to the data type afterwards. If we do any changes and re-generate the classes all manual changes will be lost.
2. After manually extending the classes we have to re-compile them. This can take a long time if we are working with big projects.

You see it would be great if we could separate the functionality from the (abstract) syntax of the data types. And this is exactly what the "visitor pattern" does. We explain the usage of this mighty design pattern with a example. We define a recursive data type list of `int` values. A list can be empty or contain a head (an `int`) and a tail (a list). vcg specification looks like this.




We generate classes with "visitor" option checked (fig. 8). vcg creates 6 JAVA files `List.java, Cons.java, Empty.java, SyntaxNode.java, Visitor.java` and `VisitorAdaptor.java`. `List, Cons, Empty` look more or less like we know it from previous examples. The difference is that they all implement `SyntaxNode` (directly or indirectly) and they have some additionally methods. The most important method is `accept(Visitor visitor)`. Its function will be described in the following examples.

The other 3 classes (or interfaces) have the following functions.

*SyntaxNode* This interface is implemented by all data type classes (the syntax nodes). It defines all important methods.

*Visitor* This interface defines the `visit()` methods for the different data types. In our example it looks like that

```
public interface Visitor {

  public void visit(List list);
  public void visit(Cons cons);
  public void visit(Empty empty);

}
```

*VisitorAdaptor* This class is a convenience class which implements `Visitor`. With extending this class for our visitors we have the advantage of defining `visit()` methods only for the nodes where we have to do something.

Confused? We'll give detailed explanation with this step by step example. The `list` data type can store a list of `int` values. Suppose we want to sum all the elements of a list. Because we do not want to alter the classes itself we use a visitor. Therefore we extend `VisitorAdaptor`.

```
public class ListSum extends VisitorAdaptor{
private int sum=0;              //this int holds the current value

  public int getSum(){          //get() method for the value
    return sum;
  }

  public void visit (Cons c){   //visit() method is defined in Visitor.java
    sum+=c.getHead();           //add the head of a list to the current value
    c.getTail().accept(this);   //"recursive" call to rest of the list
  }
}
```

Before explanation we first show usage of this visitor.

```
public class Main{

  public static void main(String args[]){

    List l=new Cons(10, new                          //set up the example list
      Cons(20, new Cons(30, new Empty())));          //"10, 20, 30"
    ListSum ls=new ListSum();                        //create our visitor

    l.accept(ls);                                    //let our visitor visit
                                                     //the list
    System.out.println("Sum is: "+ls.getSum());  //print result
  }
}
```

As we expected we get the result "60". What happened? We start the visitor with `l.accept(ls)`. The `accept()` method is (abstractly) defined by `List` without a method body. So we take a look at the `accept()` method of `Cons`.

```
public void accept(Visitor visitor){
  visitor.visit(this);
}
```

`accept()` only calls the `visit()` method of our visitor, which in turn adds the head value to its variable `sum`. After adding the value it calls `accept(ourVisitor)` for the rest of the list. Ok, but the at the end of our list is an instance of `Empty`. Our visitor only defines `visit(Cons c)`. We take a look at the super class of our visitor, `VisitorAdaptor` and see it defines `visit(Empty e)`. The method has a empty body and therefore does nothing. It adds nothing to our sum and does not call to a new method. The visiting stops at this point.

We added functionality to our classes without extending themselves. As you'll see in the next example it's easy to write another visitor which produces a string concatenation of all values of a list. This visitor has another addition, it has a explicitly defined method `visit(Empty e)` which produces a "E" to mark the end of the list.

```
public class ListConcat extends VisitorAdaptor{
  private String concat="";              //the string holds the current value
  public String getConcat(){             //get() method for the string
    return concat;
  }
```

```
  public void visit (Cons c){            //visit() method is defined in Visitor.java
    concat=concat+" "+                   //add the head of a list to the current string
      String.valueOf(c.getHead())+" ";
    c.getTail().accept(this);            //"recursive" call to rest of the list

  }
  public void visit (Empty e){           //visit() for empty list
    concat=concat+" E ";                 //add a "E"
  }                                      //do not call further methods
}
```

We now use both visitors.

```
public class Main{

  public static void main(String args[]){

    List l=new Cons(10, new                       //set up the example list
      Cons(20, new Cons(30, new Empty())));       //"10, 20, 30"
    ListSum ls=new ListSum();                     //create our sum-visitor
    ListConcat lc =new ListConcat();              //create our concat-visitor

    l.accept(ls);                                 //let our visitors visit
    l.accept(lc);                                 //the list
    System.out.println("Sum is: "+ls.getSum());  //print result
    System.out.println("Concatenation is: "+
      lc.getConcat());
  }
}
```

We get the following result:

```
Sum is: 60
Concatenation is:  10  20  30  E
```

In some cases (e.g. the concatenation visitor) the order in which the nodes are visited is important. vcg generates methods to visit the nodes in “top-down” or “bottom-up” manner. To use this methods we have to do a small change to our visitor. We don’t have to call other `accept()` methods. The methods `traverseTopDown()` and `traverseBottomUp()` do this for us.

```
public class ListConcat extends VisitorAdaptor{
  private String concat="";               //the string holds the current value
  public String getConcat(){              //get() method for the string
    return concat;
  }

  public void visit (Cons c){            //visit() method is defined in Visitor.java
    concat=concat+" "+                   //add the head of a list to the current string
      String.valueOf(c.getHead())+" ";   //NO call to other accept() methods
  }

  public void visit (Empty e){           //visit() for empty list
    concat=concat+" E ";                 //add a "E"
  }                                      //do not call further methods
}
```

You can see the difference between top-down and bottom-up traversing in the example.

```
public class Main{
```

```
  public static void main(String args[])
  {
    List l=new Cons(10, new Cons(20, new Cons(30, new Empty())));

    ListConcat lc= new ListConcat();
    l.traverseTopDown(lc);
    System.out.println("Concatenation 1 is: "+lc.getConcat());

    lc = new ListConcat();
    l.traverseBottomUp(lc);
    System.out.println("Concatenation 2 is: "+lc.getConcat());
  }
}
```

We have to create a new visitor for the second traverse since our visitor has no method to reset its variable which holds the result. Running the program produces:

```
Concatenation 1 is:  10  20  30  E
Concatenation 2 is:  E  30  20  10
```

These small examples should have shown the usage and might of the “visitor pattern”. For further information refer to [3].

# 3 The vcg user interface

The vcg window is divided in two parts. On the left is the specification editor, here you can create new or edit existing specifications. On the left is the output window with three tabs to switch between its three different modes. The first mode is the diagram representation (3.3.1) where the data types a represented with the tree-like structure introduced in the last section. The second mode shows a class tree and third mode reports errors. The separator between the specification editor and the output window is one-touch-expandable. By clicking on of its little arrows you can enlarge or shrink the output window and/or specification editor.

The vcg user interface has a list of features we like to introduce here.

*Cut&paste* The specification editor is simple but has all important editor features. It supports cut&paste to and from clipboard on all JAVA-enabled platforms.

*Syntax highlighting* The editor has syntax highlighting capabilities. It highlights keywords blue, idents red and comments green. By supporting your perception syntax highlighting greatly improves editing efficiency.

*Instant feedback* vcg is multi-threaded and uses a second thread to auto-update the output window. You just have to stop typing for short time and vcg issues a re-parse of the specification. No need to press the “parse” button after every modification. If you feel disturb by this function, just hide the output window with the one-touch-expandable separator.

*Tree-like representation* For a clearer representation vcg uses a tree-like structure to display the data types. Due to vcg’s layout algorithm you can handle even large specification without loosing track of your data types.

*Direct manipulation* The data type representation supports direct manipulation, so you can modify the graph if you’re are not fully satisfied with the auto-layout.

*Diagram export* vcg supports a JPEG-export of the current specification. You can use this for documentation or teaching.

*Online help* vcg has a built-in HTML-based online-help.

*Error messages* Error messages are visualized in a messages pane. The messages pane gives a description of the error and holds information on line of occurrence.

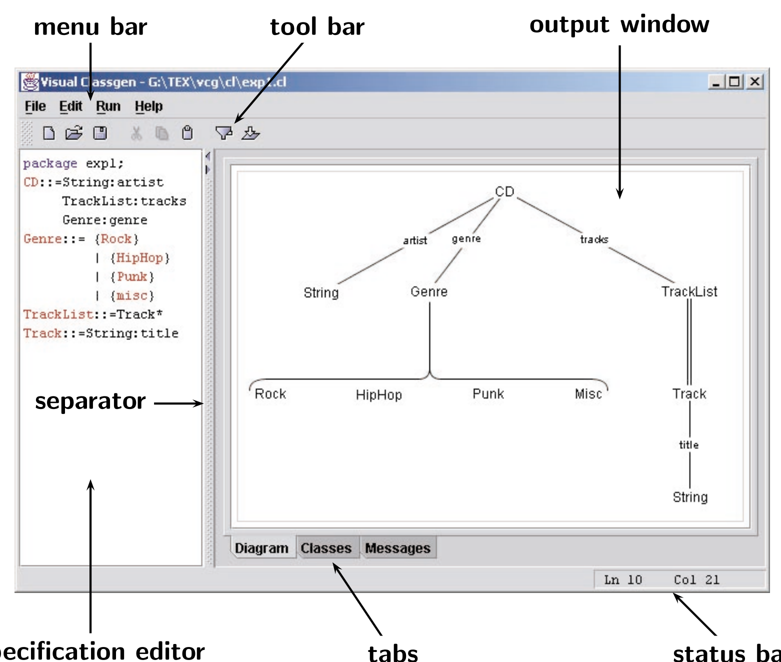



Figure 7: vcg main screen

## 3.1 Pull down menus and tool bar

The menus and the tool bar give access to standard file and editor operations. Every tool bar button has a corresponding menu entry.

### 3.1.1 Menu bar

The menu bar contains four pull down menus: "File", "Edit", "Run" and "Help". They contain the following commands:

**File menu** The "File" menu is responsible for all file operations concerning the specification. It has a last of the most recently used specifications.

*New* Clears editor and output window. If there is an unsaved specification, you'll be asked to save.

*Open...* Opens a file chooser dialog to select a specification.

*Save* Saves the current specification. If it is unnamed, "Save as..." is called instead.

*Save as...* Open a file chooser dialog to save the current specification under a different name.

*Export diagram...* This command allows to export the current diagram as a JPEG-file. I opens a file chooser dialog to specify the file name. The whole diagram is exported. There's currently no possibility to export selected parts only.

*Exit* Quits program. If there is an unsaved specification, you'll be asked to save.

**Edit menu** The "Edit" menu contains command which support cut&paste of the specification editor.

*Cut* Cuts selected text to clipboard.

*Copy* Copys selected text to clipboard.

*Paste* Pastes from clipboard.

**Run menu** "Run" menu offers commands to start parsing of the specification and to generate the JAVA classes.

*Parse* Starts parsing of the specification and updates diagram, class tree and messages. Due to vcg's multi-threaded design you'll hardly use this command, since parsing is started automatically if you pause during typing. You can use this command to re-layout the graph.

*Generate classes* Opens the "Generate classes" dialog (fig. 8). Here you can specify the output directory for the classes. If you uses the `package` (2.1.1) command vcg creates new directories for the package. Additionally you can set the parameters "visitor" and "overwrite". If "visitor" is checked vcg generates classes with support for the visitor pattern (2.5). "overwrite" forces vcg to overwrite existing classes without asking. After class generation the dialog displays a status report.

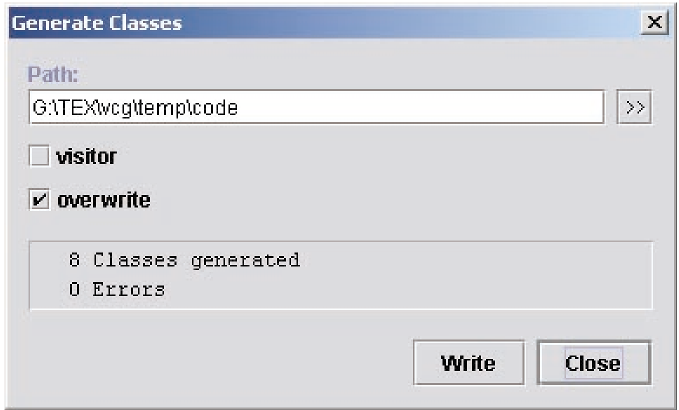



Figure 8: "Generate classes" dialog

#### Help menu

*About* Opens the about box.

*Help* Opens the HTML-based online help (3.4).

### 3.1.2 Tool bar

All tool bar commands have a corresponding command in the pull down menus. You can use them for faster access to often used commands. The tool bar is dockable, you can move it to every desired place.

- Clears editor and output window. If there is an unsaved specification, you'll be asked to save.
- Opens a file chooser dialog to select a specification.
- Saves the current specification. If it is unnamed, "Save as..." is called instead.
- Cuts selected text to clipboard.
- Copys selected text to clipboard.
- Pastes from clipboard.
- Starts parsing of the specification and updates diagram, class tree and messages. Due to vcg's multi-threaded design you'll hardly use this command, since parsing is started automatically if you pause during typing. You can use this command to re-layout the graph.
- Opens the "Generate classes" dialog. See "Run" menu for further description.

## 3.2 Specification editor

The specification editor works like a normal text editor. It supports cut&paste vie menu or tool bar commands. Additionally the specification editor has syntax highlighting capabilities. Keywords are highlighted blue, idents red and comments green. During specification editing a second thread watches you're typing and issues a re-parse of the specification if you stop for a certain time. Therefore you have instant feed-back in the output window.

## 3.3 Output window

The output window has three tabs a the bottom (fig. 7) to switch between its three instances "diagram", "class tree" and "messages".

### 3.3.1 Diagram pane

The diagram pane displays the data types in the tree-like structure introduced in previous sections. The representation is not static, you can drag and drop all nodes (since the tree-like representation, we speak of nodes). With a single mouse click you can select a node and its whole sub-tree. You can now drag the sub-tree to the desired place. If you want to select more than one node you can use the rect-selector tool. You just have to click and hold the mouse button down, if you move the mouse now you'll see a grey rectangle to select the nodes. You can also use this tool to select a single node without its sub-tree. Furthermore the selectors are click sensitive. You can select and drag them. If you did any modifications to the diagram and like to export it, you'll have to be careful not issue a re-parse. Parsing resets the diagram to its original form.

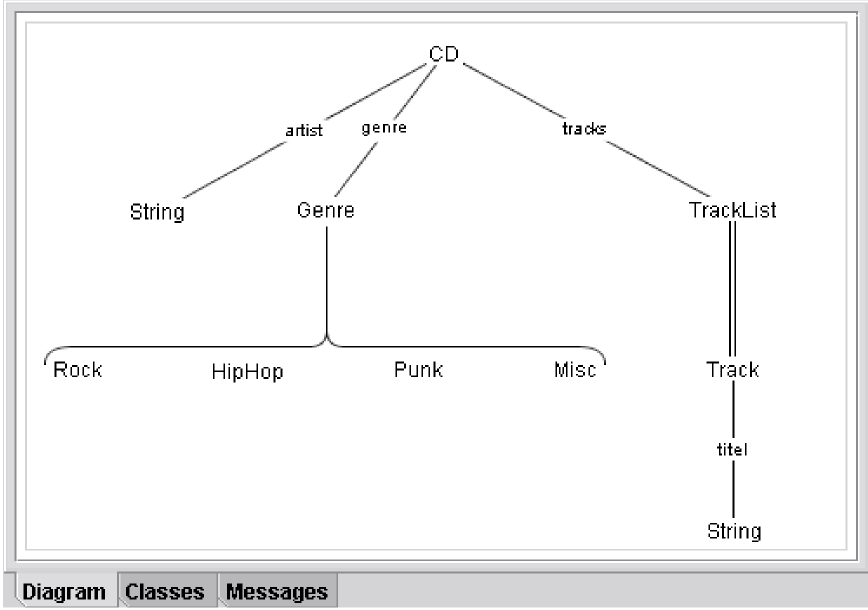
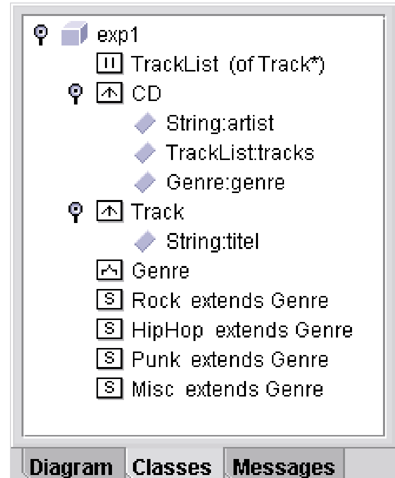



Figure 9: the diagram pane

### 3.3.2 Class tree

vcg offers another representation of the specification, the class tree (fig. on the right). The class tree lists all data types which are visualized in the diagram. Additionally it contains enumerations (2.2.1) and attributes (2.2.2). The class tree uses the following icons for the different data types.

- The package.
- A record.
- A list.
- The base class of a variant.
- A sub class of a variant.
- An enumeration.
- A element of a record.
- A attribute.




### 3.3.3 Messages

The third instance of the output window is the messages pane. The message pane reports errors (fig. 10) with description and line number. The status bar (fig. 7) gives information about the current line number. After generation of java classes it contains logged information about the generation (fig. 11).

## 3.4 Online help

The vcg online help (fig. 12) is HTML-based and not context sensitive. You can use it like a very simple web browser. On the left is a table of contents pane for faster navigation.

Figure 10: error messages

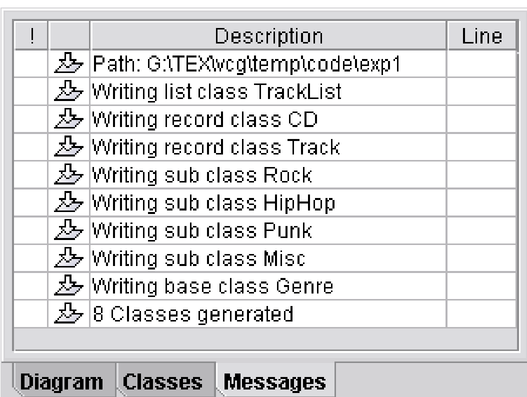



Figure 11: the generation log

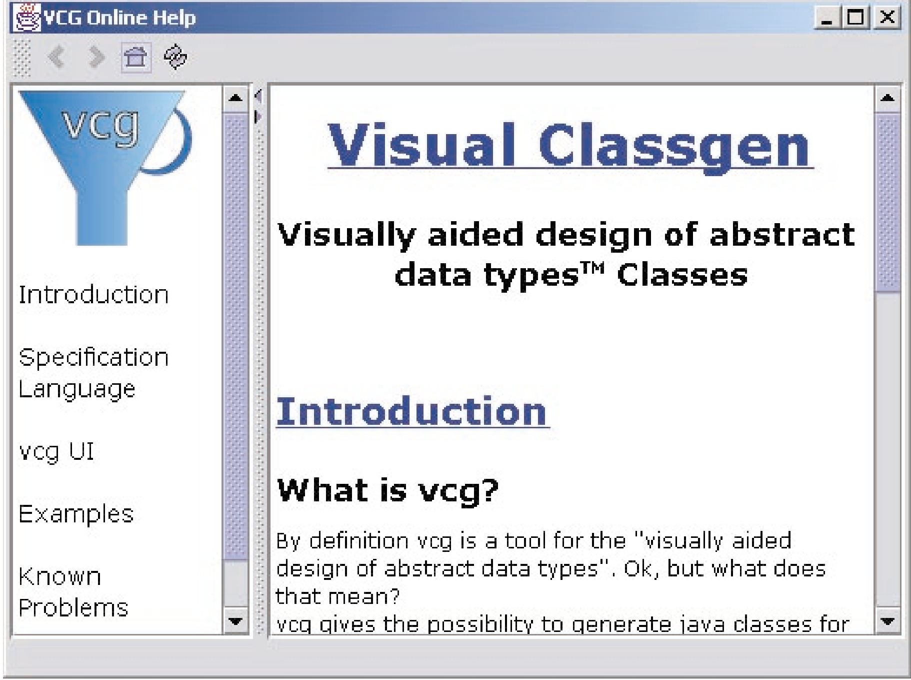



Figure 12: the online help

# 4 Examples

Here are three bigger specification examples.

## 4.1 vcg syntax

This example describes the syntax of vcg specification language.

```
// classgen syntax nodes

package classgen.syntax;

attr StringList nonTerminals, classes with Specification;
attr String     extending             with Production, Item;
attr "boolean"  generate              with Alternative;


Specification   ::= String:name AttribDecl*:attributes Production*:productions Method*

Production      ::= {GrammarProduction}
                    String          : name
                    Alternative*    : alternatives
                    AttribDecl*     : attributes
                    "Boolean"       : isList
                    "boolean"       : hasAlternatives
                  | {EnumProduction}
                    String          : name
                    String*         : elements

Alternative     ::= String:name Item*:items

Item            ::= {RecordItem} String:ident String:selector "int":line
                  | {ListItem}   String:itemType String:ident String:selector "int":line

AttribDecl      ::= Integer:specifier String:type StringList:selectors StringList:withList

Specifier       ::= one of "SYNTHESIZED", "INHERITED", "ATTRIBUTE"

Method          ::= String:className Line*:lines

Line            ::= "int":line String:text


methods of Alternative {
  public static void main(String argv[]) {
    System.out.println("hello");
  }
}
```

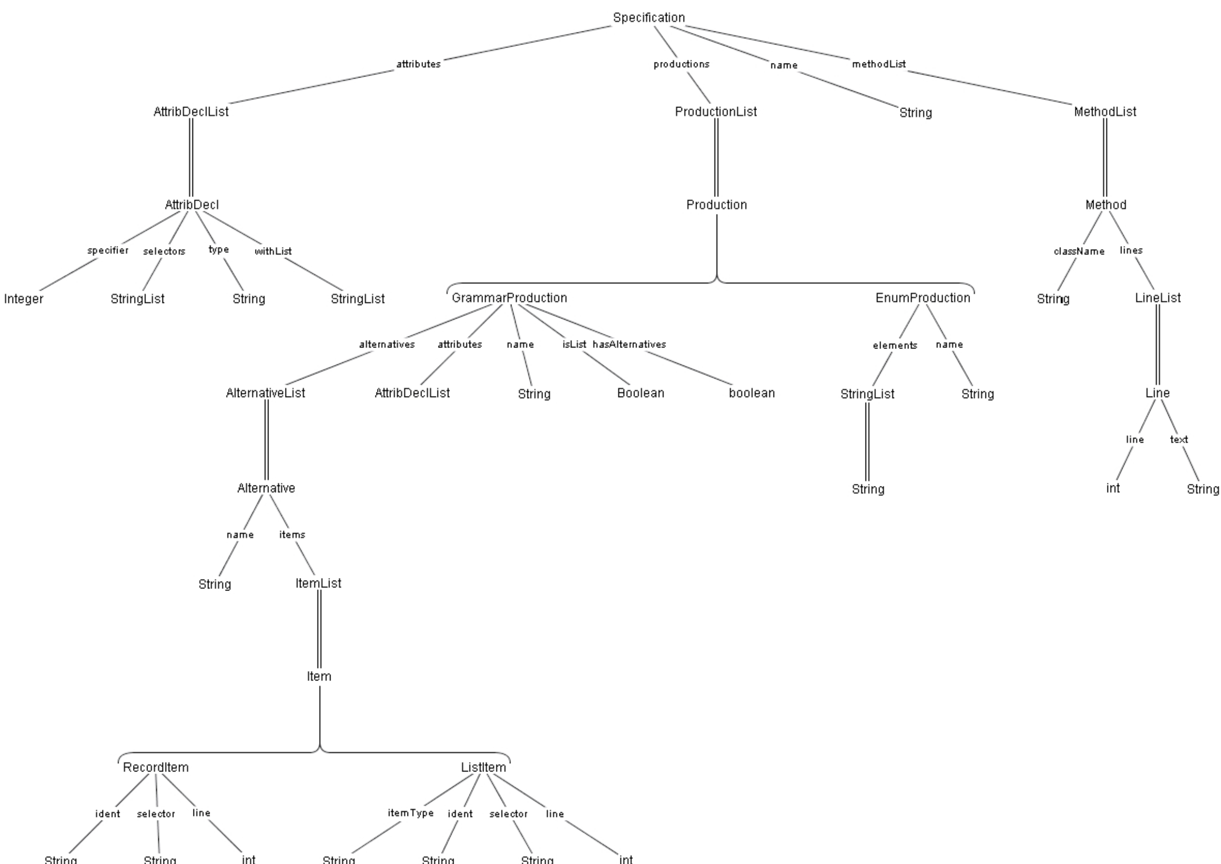



Figure 13: vcg syntax

## 4.2 Spreadsheet

This specification was used for a small spreadsheet application.

```
attr Cell       link           with CellPos;
attr Value      evalResult     with Formula;
attr "boolean"  calculated     with Formula;

Spreadsheet     ::= Column*
Column          ::= Cell*
Cell            ::= Formula     : formula

Formula         ::= {ValueFormula}
                        Value   : value
                  | {CellPosFormula}
                        CellPos : cellPos
                  | {BinFormula}
                        Formula : lhs
                        Operator: op
                        Formula : rhs

Operator        ::= {Plus} | {Minus} | {Times} | {Div}

Value           ::= {StringValue}
                        String  : stringValue
                  | {DoubleValue}
                        "double": doubleValue
                  | {EvalError}
                        String  : message

CellPos         ::= {AbsPos}
                        "int"   : lineNo
                        "int"   : columnNo
                  | {RelPos}
                        "int"   : lineDiff
                        "int"   : columnDiff
```

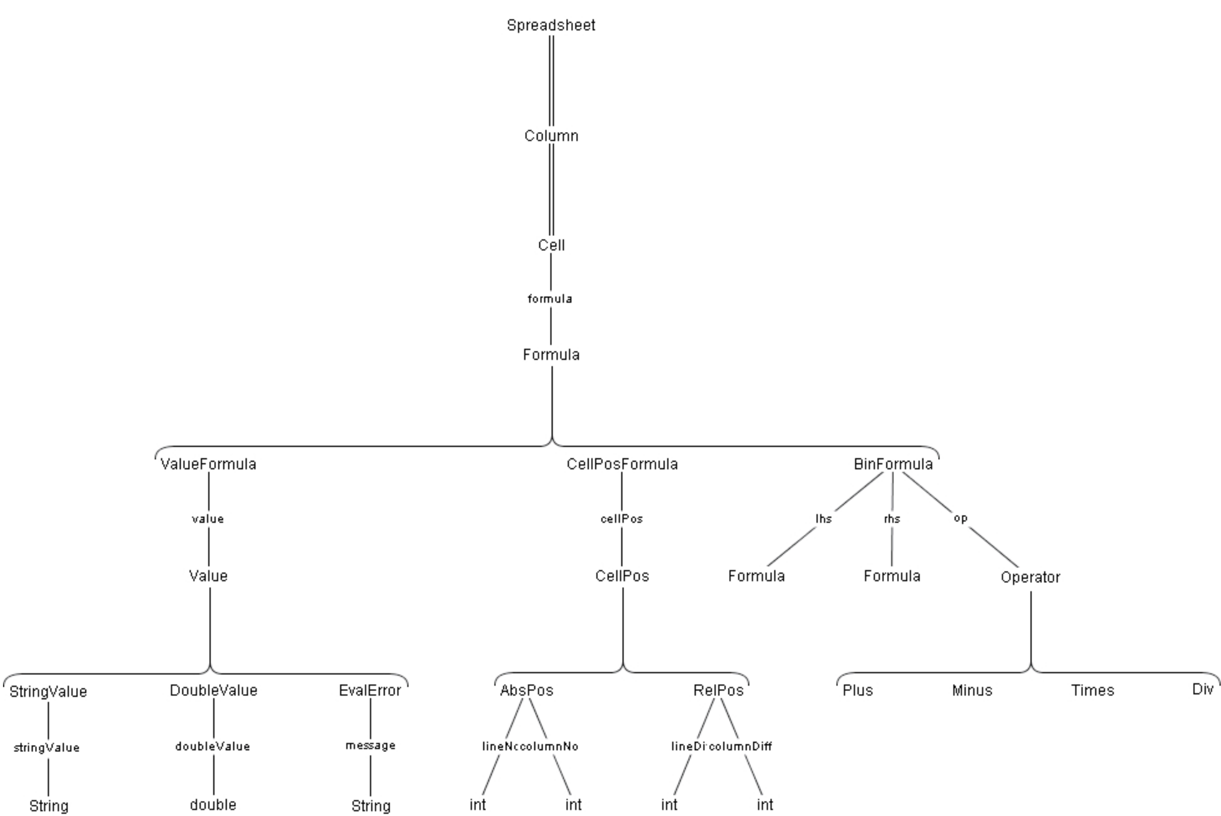



Figure 14: Spreadsheet

## 4.3 Compiler

This is a example from the compiler design lecture.

```
attr "boolean" istvarParam, istformParam with Dekl;
attr DeklInfo deklinfo with Id_Seq, Var;

Prog ::= "String":ident  Proc_Dekl_Seq  Block


Proc_Dekl_Seq ::= Proc_Dekl Proc_Dekl_Seq
                | /* empty list */

Proc_Dekl     ::= "String":ident Form_Par_Seq Block

Form_Par_Seq ::= Form_Par Form_Par_Seq
               | /* empty list */

Form_Par     ::= { Param }     Var_Dekl
               |  { Var_Param } Var_Dekl

Var_Dekl     ::= { Variable }  Typ  Id_Seq:id_Seq
               |  { Array_Variable } "int":laenge Typ Id_Seq:id_Seq

Typ      ::=    "int":typ

Block    ::= Dekl_Seq  Anw_Seq

Dekl_Seq ::= Dekl Dekl_Seq
           | /* empty list */

Dekl     ::= { Var_Dekl }


Id_Seq   ::= "String":ident Id_Seq
           | "String":ident


Anw_Seq ::= Anw Anw_Seq
          | Anw

Anw     ::= Ausdr


Ausdr     ::= {BinAusdr}  Ausdr "int":op Ausdr
            | {IntAusdr}  "int":value
            | {VarAusdr}  Var

Var        ::= {IdentAusdr}   "String":ident
             |  {ArrayAusdr}   "String":ident Ausdr:index
```

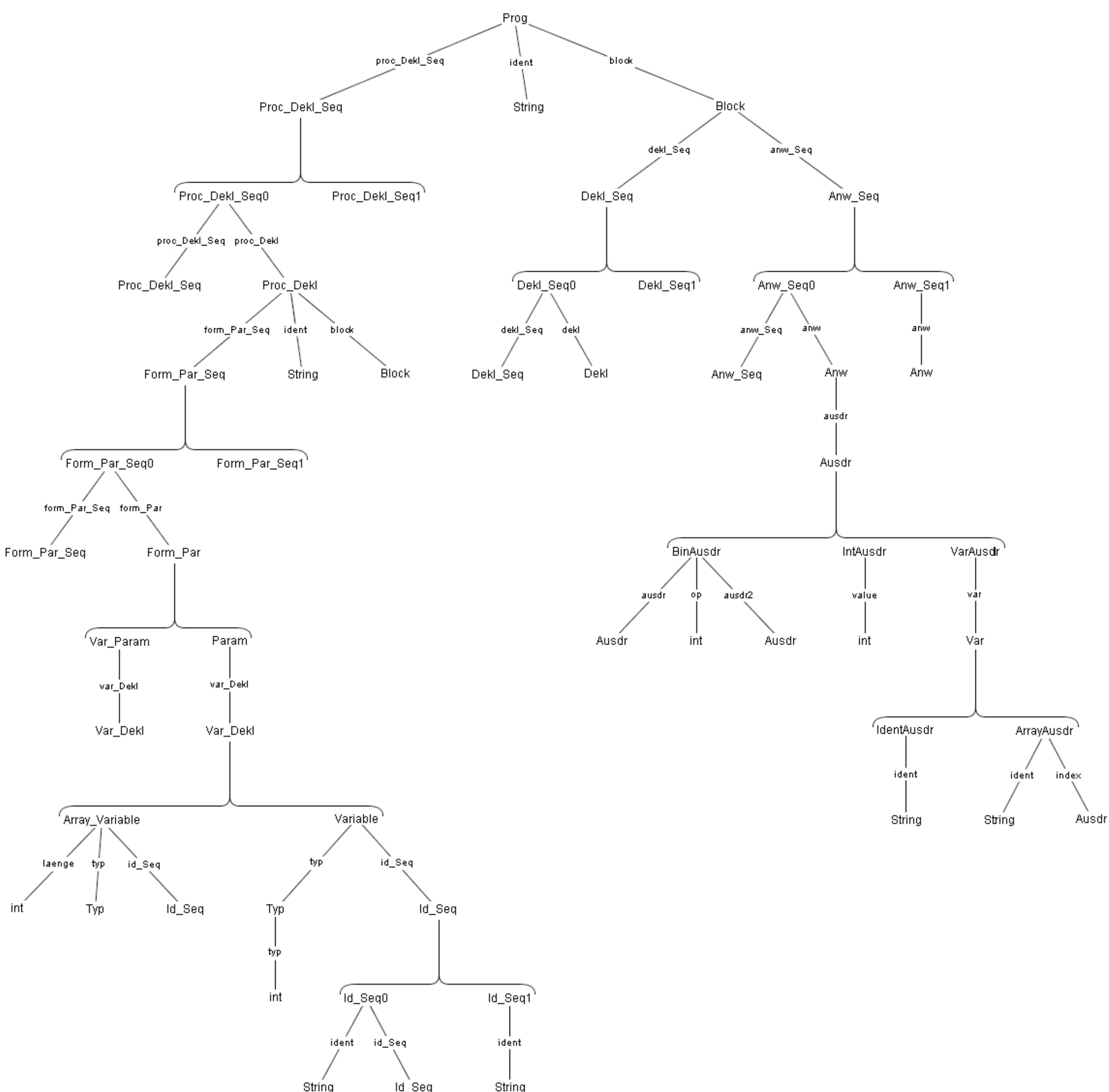



Figure 15: Example 3

## 5 Known problems

There are a few small problems in this release of vcg. Most of them have to do with error handling and the messages pane.

- Sometimes all messages are deleted and there is no other status information. This can occur after a syntax error was recognized. Please complete the current production or undo what you type before. The messages will be back again.
- The current version has to deal with a problem in the JAVA CUP parser. Sometimes you'll receive the error message "Symbol recycling detected (fix your scanner)". Please don't worry about this message. Most likely there's no error in your specification. Probably there is a uncompleted production.
- After generation of classes the generation dialog display a status message like "4 classes generated". Additionally the messages pane lists the classes. Unfortunately not all classes are counted and listed. Status information and messages pane omit enumerations and visitor pattern related classes.

## References

[1] G. Klein: *classgen.* 1999

[2] S. Winter and F. Deissenböck: *Visual Classgen developers documentation.* System development project. Technische Universität München, 2001

[3] E. Gamma, R. Helm, R. Johnson and J. Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994

# Index

alternative, *see* variant
attribute, **10**

brace, 12

class tree, 10, 17, **20**
`classgen`, 3, 6
compiler design, 26
constructor, 11
cut&paste, 17

diagram, 5, 17
diagram export, 17

edit menu, 18
element, 6, 8
Enumeration, **10**
example
  visitor pattern, 13
examples
  Box, 8
  CD, 4

file menu, 18

generation dialog, **18**
get/set-methods, 7
grammar, 3, **6**

help menu, 19

instance variables, 7
instant feedback, 17

JPEG, 3

list, 5, **9**

messages pane, 3, 20
method, **11**
`methods of`, 11

`one of`, 10
online help, 20
output window, 3, **17**

`package`, 6
production, 6, 12

record, 4, **6**
rect-selector, 19
run menu, 18

selector, 4, **7**
separator, 17
specification, **3**
  syntax, **12**
specification editor, 3, **17**
spreadsheet, 24
symbol recycling, 28
syntax highlighting, 17, **19**

tabs, 17, 19
tool bar, 19

variant, 4, **8**, 12
  base class, 4
  sub class, 4
`Vector`, 9
visitor pattern, 3, **13**
  `traverseBottomUp()`, 15
  `traverseTopDown()`, 15
  `SyntaxNode`, 13
  `VisitorAdaptor`, 14
  `Visitor`, 13