Lava – An Object-Oriented RAD Language Designed for Ease of Learning, Use, and Program Comprehension

Klaus D. Günther[1], Irmtraut Günther[1]

[1] GMD, Institute for Secure Telecooperation, Rheinstr. 75,
D-64295 Darmstadt, Germany
{Klaus.Guenther, Irmtraut.Guenther}@darmstadt.gmd.de

Abstract. The growing demand for new software calls for a considerable acceleration of the software production process and for a sensible relaxation at the software maintenance front. These goals can be achieved only if we can decisively increase the degree of modularity, variability, comprehensibility of software, or short: the degree of structured programming, as well as the simplicity of program manipulation, restructuring, and transformation. The experimental object-oriented language "Lava" and the associated programming environment "LavaPE" attempt to achieve these goals by providing quite a number of unusual features. The most remarkable features are: 1. Text editors are completely replaced with Lava-specific structure editors. 2. A Lava class consists of a public "interface" and a completely separate, exchangeable "implementation" which may be stored in a different file. 3. Frameworks and design patterns are supported in a very natural way by allowing packages and interfaces to have overridable type parameters.

1 Introduction

The continuously growing demand for new software can be supplied only if the productivity of the programming process can be decisively increased. (Cf. section 3 of the PITAC Report [10] to the American government, which assigns maximum priority to this goal.)

Moreover, the amount of work flowing into the continuous maintenance of large commercial software products throughout their life cycle can be decisively reduced if programming languages and programming environments encourage or even enforce (to some degree) a clear and natural subdivision of programs into small, self-contained, self-evident, independently comprehensible units.

All this means that we cannot be satisfied with the popular programming languages like C++ [11], Java [3], or Visual Basic. Rather, we have to aim for greatly improved languages and RAD programming environments

- that relieve the programmers from clerical and error-prone work,
- that in particular replace text-editors with several kinds of structure-editors,
- that are easy to learn and get along with a minimum of orthogonal concepts,
- that facilitate program comprehension,
- that facilitate clear syntactic separation of abstraction levels
  1. in the small, by favoring small, self-contained (possibly recursive) functions rather than complicated, deeply nested loop constructs,
  2. in the large, (a) by supporting nesting of declarations according to their primary or auxiliary nature, (b) by strictly separating interfaces from implementations, (c) by utilizing multiple inheritance to compose large classes in an easily configurable way from specialized small base classes that can be implemented independently,
- that clarify the control flow as well as the data flow of programs by appropriate constructs and restrictions,
- that unify what should not be separated ("embedded SQL"),
- that separate what should not be intermingled (interfaces and implementations),
- and that support reuse and multiple versions of certified components and proven design patterns from clearly documented component and pattern libraries.

The experimental programming language Lava and the associated LavaPE programming environment offer solutions to quite a number of these problems. The open-source project Lava is intended primarily to provide a public playground for experimenting with new ways of combining advanced object-oriented language features with ease of use and comprehension. Everybody is invited to participate in this explorative process. Lava is particularly attractive for those researchers who are interested in program analysis, program synthesis, and program transformation, since Lava programs are processed internally as “abstract syntax trees” all the time, beginning from their construction in specific structure editors until their execution by the Lava interpreter.

An early preview version of the Lava programming environment LavaPE (for Windows 9X/NT/2000 platforms, including a few code samples) can be downloaded from the Lava web site [8]. There you can find a more comprehensive online documentation and further papers. Cf. also [4].

Section 2 below deals with Lava contributions to the goals "ease of learning" and "ease of use". Section 3 explains how program comprehension is facilitated by quite a number of different Lava features. Section 4 outlines how Lava copes with "genericity" in a new way, based on “virtual types". These are particularly suited to provide very natural specifications of "design patterns" and "frameworks", viewed as (groups of mutually recursive) “virtualized” types. This capability will play a much more important role in future languages.

2 LavaPE and Ease of Learning / Ease of Use

2.1 Replacing Text Editors with Structure Editors

Lava programs are no longer "written" but "constructed" from basic constructs, using point/click/drag/drop/cut/copy/paste and menu selection operations, and this is true also for the executable parts of Lava programs.

The Lava programming environment LavaPE is completely based on structure editing, with two dominating, primary views: the "declaration view" and the "exec view" (see Fig.1, last page):

- The declaration view is used for declaring various kinds of Lava entities, in particular new packages, interfaces, implementations, and their respective substructures.
- The exec view is used to construct the executable portions of Lava programs, i.e.,
  1. "exec's", = bodies of functions and of "initiators" (= main programs),
  2. "constraints", which may be associated with interfaces and must be fulfilled whenever a new object supporting the respective interface has been created.

The *declaration view* is a "tree view" to which everybody is accustomed, for instance from the "Explorer" of Microsoft Windows. Declarations may be nested to any depth in Lava. Tree construction is controlled by tool buttons corresponding to the basic Lava notions, like "new package", "new interface", "new implementation", "new member variable", "new member function", "new function parameter", etc. The properties of these entities are edited using appropriate property sheets. Subtrees can be easily copied and moved by drag-and-drop operations or expanded/collapsed by specific tool buttons.

There are several auxiliary tree views the most important of which is used for specifying the details of an interface or package/pattern derivation leading to a derived interface or package/pattern. (See Fig.1, last page.)

The *exec view* is a quite normal textual representation of an "exec" or "constraint". But although it uses the standard Windows "rich edit view", it is not editable directly as text. The executable program text is rather constructed from a number of basic statement, expression, and special constructs, which would typically contain "placeholders" (= syntactic variables) <stm>, <expr>, <var>, <type>, <func>, <set> ... for statements, expressions, references... that may be inserted in these places.

In fact, there is no fixed textual syntax of Lava at all, nor is there a Lava parser or compiler. The point-and click operations of the programmer generate and manipulate an internal tree representation of the Lava program (an “AST”, short for “abstract syntax tree”) *directly*. The readable representation of declarations, execs, and constraints is generated only on the fly as long as a corresponding declaration or exec view is open.

No text entry whatsoever is required in LavaPE, except for comments, constants, and new identifiers. Syntax errors cannot occur any longer. Context-related errors are reported at the earliest possible time. References to be inserted are selected from specific combo-boxes whose current content depends largely on the current selection.

So Lava is not a conventional textual language, but it is inseparably connected with LavaPE.

2.2 Automatic Maintenance of References

Readable, textual identifiers are, in a sense, meaningless in Lava. Every Lava entity has a unique internal identifier that is never changed. All references to Lava entities are based solely on these internal identifiers. A readable textual name is associated with such an immutable, unique, internal identifier at the place where the respective Lava entity is declared, and it can be changed only there.

Since readable representations of Lava programs are generated only on the fly when they are opened in one of the Lava structure editors, all references to Lava entities will always be up to date: The textual name of a Lava entity is always "fetched" from its actual declaration and inserted at the place of reference. Textual names need not be unique even. But Lava tries to prevent you from using duplicate names in Lava, of course, since they impair the comprehensibility of programs and force you to use the "go to declaration" function of LavaPE to find the actual meaning of the respective name.

Automatic maintenance of references is a quite important advantage for source code maintenance. It happens very often that you would like to assign a more meaningful name to an existing entity, but it is extremely laborious and boring to identify all affected source files and to use string search and replacement in order to change all affected references. This will often prevent you from introducing more significant names. In Lava you need only change the name in the declaration of the respective Lava entity.

Automatic maintenance of references applies also in cases where declarations are moved (using drag-and-drop) within the Lava declaration tree or even between different files: The path-names of Lava interfaces, packages, functions, etc., that reflect the position of these entities in the declaration tree, are changed accordingly in all references to the affected entities. Cf. [1] for an alternative approach to identifier change and maintenance.

2.3 Automatic Maintenance of Function Calls

Another kind of automatic maintenance of references applies to member functions of interfaces and implementations. If you add or delete or permute formal parameters of a function then all existing invocations of these functions are changed immediately or as soon as the containing Lava file is opened: Placeholders for actual parameters are inserted where new formal parameters have been inserted; actual parameters corresponding to deleted formal parameters are deleted, likewise; the order of actual parameters is adapted to the new order of the permuted formal parameters.

This is again made possible by the fact that formal parameters of functions, like all other Lava entities, have unique internal identifiers and actual parameters refer to these internally.

3 Facilitating Program Comprehension

3.1 Synoptic Declaration Trees

In section 2.1 we have outlined the nested, tree-like structure of Lava declarations and the associated structure editor. This way to deal with declarations offers decisive advantages for program comprehension:

- You need not put all declarations on a single level but you can nest them according to their primary or subordinate, auxiliary nature.
- You can expand and collapse entire subtrees and in this way switch easily between nested details and "bird's eye view", just as you need.
- You can easily navigate forth and back between declarations and references by clicking the tool buttons "go to declaration" (or double-clicking the reference) and "return to reference".
- You can easily re-arrange the tree structure by applying drag-and-drop or cut/copy/paste operations to individual tree items or to entire subtrees.

As for the “static” nested classes of Java, nesting of declarations does *not* establish a special semantic relationship between inner and outer declarations but is *only* a means to arrange primary and auxiliary declarations in a meaningful way. Inner declarations can always be referenced also from outside, unless they are nested in an *implementation*, or in a package that has been declared *opaque* explicitly.

3.2 Earlier and More Complete Error Reporting

Lava has no compilation phase but checks for errors after every individual structure editing operation. So errors are detected and reported "in embryo", and errors in executable code are highlighted by displaying the erroneous construct (mostly a single identifier or constant, rather than just an entire line of code) in boldface and red color. Likewise, placeholders that have not yet been replaced with concrete constructs in executable code are displayed in red font.

Erroneous declarations are highlighted by a small red rectangle that is affixed to the right side of the corresponding declaration icon. Error messages belonging to the current selection are displayed in a separate error window (for declarations as well as for executable code).

So you have just to look for remaining red flags in declarations and for red portions of executable code in order to figure out where your program calls for correction of errors or for completion. To this end, you can move the current selection to the next or preceding error in the declaration tree view as well as in the exec view.

Moreover, comprehension of still incomplete and erroneous programs is greatly facilitated in Lava by providing several additional features that allow us to perform more complete checks for semantic errors:

1. "Single-assignment" (section 3.5) prevents inadvertent reuse of the same variable within the same program branch with different meanings; violations are reported as errors; the last preceding conflicting assignment is highlighted on a button click.
2. Single-assignment, combined with a stringent phase-model of object creation, initialization, customization and use enables complete initialization checks. Incompletely initialized objects may be passed as parameters only to "initializers" (corresponding to constructors in Java/C++); they cannot be used for method calls. Initializers must initialize all non-optional member variables; violations are reported as errors.
3. The distinction between "value objects", that become immutable after initialization/customization, and "state objects", that may be changed again and again, provides another kind of redundancy, which enables valuable additional checks (cf. section 3.4).
4. Lava supports an advanced notion of "virtual types" with covariant specialization (section 4). This opens a new dimension of static type checking and early error reporting where you otherwise would have to resort to "type casts" and run time type checks in C++ and Java.

3.3 Strict Separation of Interfaces and Implementations

Older, non-object-oriented languages like Modula-2 and the original version of Ada that were based on "abstract data types", had already achieved a clean syntactic separation of "interfaces/definitions" and "implementations" according to the important "principle of information hiding", which we deem to be of vital importance for program comprehension and software maintainability / evolvability.

This clear separation has been lost again in object-oriented languages like C++ and Java. Although Java provides an interface notion, while Java classes contain the implementations of their member functions, you can still use classes to declare the types of variables. A Java interface does not have member variables but only member functions and thus is not suited for specifying a data structure together with a collection of methods that can be applied to it.

In contrast to this, Lava interfaces may contain member functions *and* member variables, and they are the *only* means to declare the types of variables.

Unlike Java classes, Lava implementations implement exactly one interface (and thus have the same name as the interface), They serve *only* for implementing their corresponding interface and they do not inherit from other implementations. They may contain private member variables and functions; these are not exposed to the outside world by the corresponding interface.

An interface may be marked as being "creatable". Then it may be used in "new" operations to specify the type of the objects to be created. It is the job of the Lava run time system to find an implementation of a given interface, as well as implementations of all direct and indirect base interfaces. On creation, a Lava object is composed from all these inherited interfaces and the associated implementations. Lava supports multiple inheritance with "virtual base classes", as you would say in C++: If a Lava

interface A inherits the same base interface B several times on several inheritance paths then an object of type A contains only one base object of type B. See Fig.1 (last page) for an example involving interfaces, implementations, and multiple inheritance.

3.4 Distinction Between State and Value Objects

One of the most unusual (and experimental) features of Lava is its distinction between "value objects", that become immutable after initialization/customization, and "state objects", that may be changed again and again. This requires some additional consideration to be invested by the programmers but we believe that this extra cost will pay off later (during program maintenance) by increased comprehensibility of the program.

It is just a big help in understanding the purpose and role of a variable if we know that it does not represent a variable state that can be changed again and again, but just a complex value (therefore "value object") that is constructed and completed once and that is never changed thereafter during the run time of the application. Moreover, as we have pointed out already in section 3.3, point 3, this distinction enables additional, valuable semantic checks.

3.5 Data Flow, Globals, Single-Assignment

Lava prevents all kinds of implicit data flow through global variables or static member variables by relinquishing these concepts and by allowing only explicit data flow through parameter passing and local variables.

Single-assignment, applied to parameters and local variables occurring in the same exec, makes sure that those variables cannot be reused in different meanings within the same branch of this exec. (See section 2.1 for an explanation of the “exec” notion.) Single-assignment has far reaching consequences. It enforces, for instance, a more standardized and regular way to deal with branching constructs:

```
set b ← true;
...
if ...
then set b ← false // error: b has already been set
#if
```

violates the single-assignment rule and would have to be replaced by

```
if ...
then set b ← false  // OK
else set b ← true   // OK
#if
```

Single-assignment excludes also traditional sequential loops that forward information from one pass of the loop to the next by explicitly assigning new values to certain variables in every pass. In Lava, the role of traditional loop constructs is taken over by logical quantifiers "exists" and "foreach" running over finite sets of objects, and by recursive functions. Existential and universal quantifiers replace search loops and exhaustive loops, respectively, whose passes are independent of each other and could be executed concurrently therefore. All other kinds of repetitive algorithms are expressed by recursive functions.

This shift of perspective away from multiple assignment and loop constructs towards a more mathematical view of algorithms, based on exclusive logical distinctions, quantifiers and recursive functions will certainly require some relearning. But it promises to lead to smaller, more self-contained functions eventually and it will greatly facilitate program comprehension if programmers learn to think in these terms.

Absence of global variables and single-assignment cause the data flow to be strictly directed from top to bottom within executable code: The data flow of programs is clarified in a similar way as the control flow has been clarified by abandoning "go to".

4 Design Patterns and Genericity

Lava allows declarations to be grouped in "packages" similar to Java packages. Lava packages are contained completely in one Lava file and are just a special type of nodes in the Lava declaration tree. Packages and interfaces may be endowed with type parameters, called "virtual types". These may be overridden in derived interfaces and packages by assigning more derived types to them. The types of member variables and method parameters may be such virtual types. Based on this virtual type notion, Lava allows you to define groups of mutually recursive interfaces with "covariant specialization" of (virtual) member and method parameter types. This is a very natural way to support reusable "design patterns" in the sense of [2] (cf. also [6], [12], [13]) and a powerful alternative to the traditional parametric types/templates of C++ [11], Eiffel [9], and the Java genericity extension GJ [5]. Another highly desirable consequence of using patterns is that "covariant specialization" renders the ubiquitous "type casts" of C++ and Java programs superfluous.

The extension of the derivation and (multiple) inheritance notions from interfaces to patterns/packages, combined with declaration nesting, is perhaps also an appropriate way to describe the derivation / descendence relations between the individual patterns of a "pattern language" [6] or at least certain aspects thereof.

5 Conclusion

Quite a number of unusual features establish the novel and experimental nature of Lava. The use of structure editors instead of text editors relieves the programmers from syntax learning and prevents syntax errors from the beginning. The synoptic tree representation of nested declarations with its collapse/expand, drag-and-drop, go-to-declaration, override support and other functions will greatly facilitate program (re)structuring and program comprehension. The distinction between immutable value and variable state objects allows us to express more semantics in Lava. The more stringent object initialization/customization discipline, the single-assignment concept, the absence of global variables and traditional sequential loop constructs will clarify the data flow and enforce more standardized program structures based on small recursive functions. Some of these features enable more detailed semantic checks and early error reporting. Advanced support of genericity by "virtual types" opens a new dimension of program structuring by reusable design patterns. It avoids ugly "type casts" and enables a higher amount of static type checking.

Although not treated in this paper: The purely declarative treatment of concurrency, synchronization and transactions and the seamlessly integrated support for database queries, based on logical conjunctions, quantifiers and a "select" expression (as a substitute for "embedded SQL") promise to greatly reduce the learning effort of database programmers and to remove the root of many potential errors.

References

1. Caprile, B., Tonella, P.: Restructuring Program Identifier Names. Proceedings IEEE ICSM'00, 2000, ISBN 0-7695-0753-0
2. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995, ISBN 0201633612
3. Gosling, J., Joy, B., and Steele, G., Bracha, G.: The Java Language Specification. Addison-Wesley, 2000, 896 pages, ISBN 0201310082
4. Günther, Klaus D.: Lava – Programmieren im Lego-Stil. Proceedings Component Developers and Users Forum 2001
5. GJ: http://www.cs.bell-labs.com/who/wadler/pizza/gj/Documents/index.html
6. Hillside Group, Pattern Home Page: http://hillside.net/patterns/
7. Java: http://www.javasoft.com
8. Lava: http://www.darmstadt.gmd.de/~guenthk/Lava/
9. Meyer, B.: Eiffel: The Language. Prentice Hall Europe, 1992, ISBN 0132479257
10. PITAC Report to the American Government: http://www.ccic.gov/ac/report/
11. Stroustrup, B.: The C++ Programming Language, Special Edition. Addison-Wesley (2000), ISBN 020170073
12. Thorup, K.K., Torgersen, M.: Unifying Genericity (Combining the Benefits of Virtual Types and Parameterized Classes). Proceedings ECOOP'99, 186-204
13. Tonella, P., Antoniol, G.: Object-Oriented Design Pattern Inference. Proceedings IEEE ICSM'99, 1999, ISBN 0-7695-0016-1

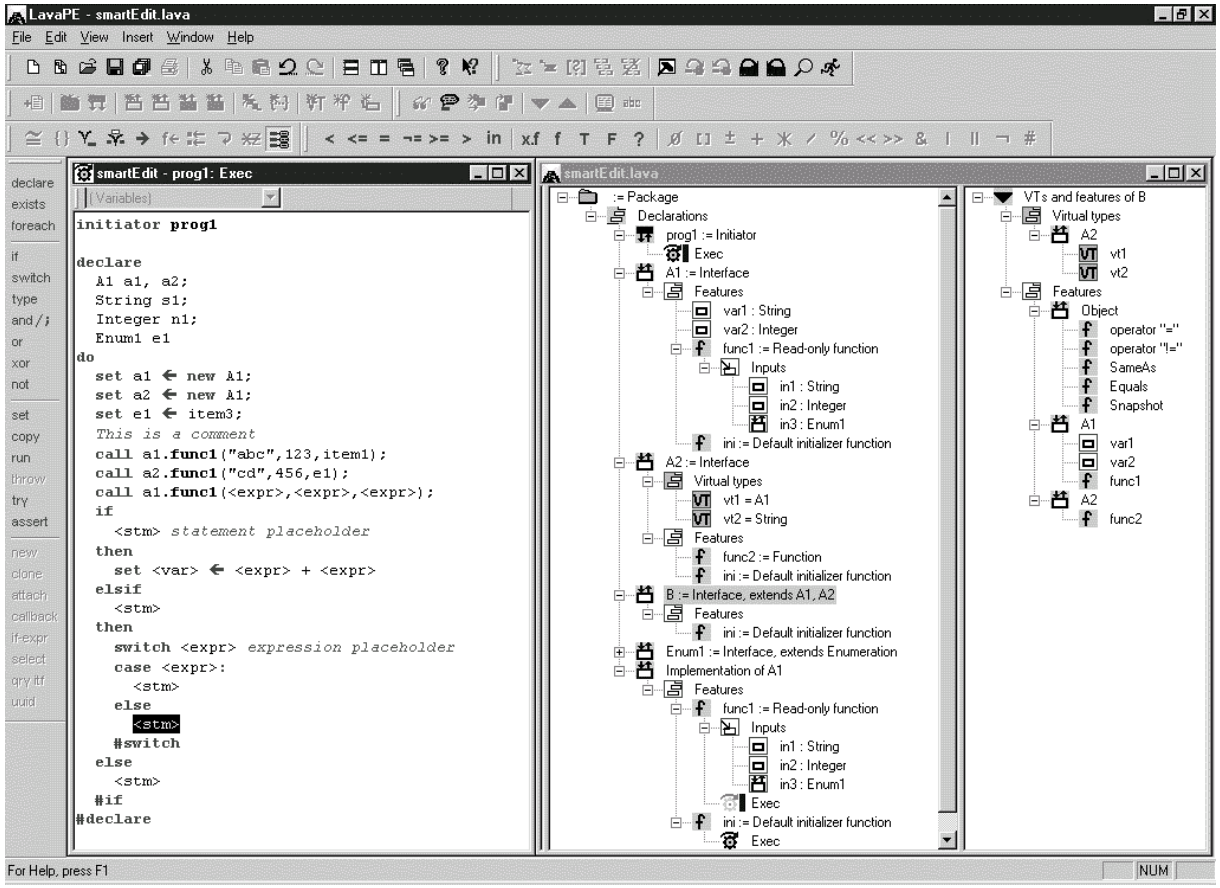

Fig. 1: Lava exec, declaration, and override view