SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; **33**:461–480 (DOI: 10.1002/spe.516)

Mechanisms for secure modular programming in Java

Lujo Bauer*,†, Andrew W. Appel and Edward W. Felten

Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08544, U.S.A.

SUMMARY

We present a new module system for Java that improves upon many of the deficiencies of the Java package system and gives the programmer more control over dynamic linking. Our module system provides explicit interfaces, multiple views of modules based on hierarchical nesting and more flexible name-space management than the Java package system. Relationships between modules are explicitly specified in module description files. We provide more control over dynamic linking by allowing import statements in module description files to require that imported modules be annotated with certain properties, which we implement by digital signatures. Our module system is compatible enough with standard Java to be implemented as a source-to-source and bytecode-to-bytecode transformation wrapped around a standard Java compiler, using a standard Java virtual machine (JVM). Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: information hiding; abstract data types; computer security; hierarchical module system; name-space management; dynamic linking; Java

1. INTRODUCTION

The traditional method of providing software-based protection within a program is by using abstract data types and information hiding. These methods have been used extensively to make sure that objects can be written in ways that allow outsiders only carefully controlled access to their implementation details.

Hiding the implementations and private interfaces of objects or classes provides for information hiding on too fine-grained a scale. We argue that the building blocks of today's object-oriented software systems, however, are not objects or classes but modules. Modules must provide a framework for information hiding and should help structure the interaction between different parts of a program. They must do this not only to protect programs from non-malicious mistakes made by other parts of the same software system, but also to protect the entire software system from malicious attack.

*Correspondence to: Lujo Bauer, Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08544, U.S.A.
†E-mail: lbauer@cs.princeton.edu

Contract/grant sponsor: National Science Foundation; contract/grant number: 9870316

Received 15 May 2002
Revised 22 August 2002
Accepted 22 August 2002

Published online 5 March 2003

The Java package system [1] is a module system, but its notions of information hiding and access control leave much to be desired, especially in hostile environments. Java packages have limited ability to control access to their member classes, they do not have explicit interfaces and they do not support multiple views of modules. These limitations make packages too weak to be used as an information-hiding mechanism.

An additional problem confronts dynamically linked programs: a piece of code is designed to behave properly only when its unresolved symbols are matched against the particular set of external objects with which the author intended his module to be linked [2]. However, since linking is often not under the control of the programmer who wrote the module—as in the Java virtual machine (JVM), for example—steps must be taken to ensure that after linking a program will behave in a manner consistent with the programmer's intentions. Type checking guarantees that the types of symbols in the interfaces between modules match, but it does nothing else to ensure that the objects with which a program links will behave in the manner that the programmer expects.

Some languages, such as Standard ML [3] with its associated Compilation Manager [4], develop the idea of module-level information hiding by providing the facility for structuring modules hierarchically. Lower levels in a module hierarchy can communicate across more expressive interfaces; higher levels can enforce more restrictive ones.

We present an ML-style hierarchical module system that improves upon Java packages by providing explicit interfaces, multiple views of modules based on hierarchical nesting and more flexible name-space management. Building on this framework, we give the programmer more control over what external modules his code can be linked with. We use digital code signing in a more meaningful way than previous approaches [5,6,7]. The details of the linking process remain abstract to the programmer and the linking specifications are simple and declarative.

The need-to-know paradigm stipulates that security is enhanced if parties only have access to the information that they need, and the information that they do not need and could perhaps misuse is hidden from them. Our module system implements this idea by allowing clients to see only a top-level interface (the information they need to know) and hiding from them the interfaces of subordinate modules (the information they do not need and might be able to misuse). In other words, only information that is explicitly deemed safe is shown to clients; hence, security is enhanced.

Some of the features of our module system can be implemented in standard Java using class loaders [8]. Our method, however, emphasizes a declarative, rather than procedural solution to the problem. A convenient declarative interface absolves the programmer of worry about the implementation details of the module-system features that he wishes to use.

Our module system is compatible enough with standard Java to be implemented as a source-to-source and bytecode-to-bytecode transformation wrapped around a standard Java compiler and a standard JVM.

2. AN EXAMPLE

Our module system, like Java packages, groups classes into larger units. A module in our system consists of a set of source files and a module description file. The module description file consists of three parts:

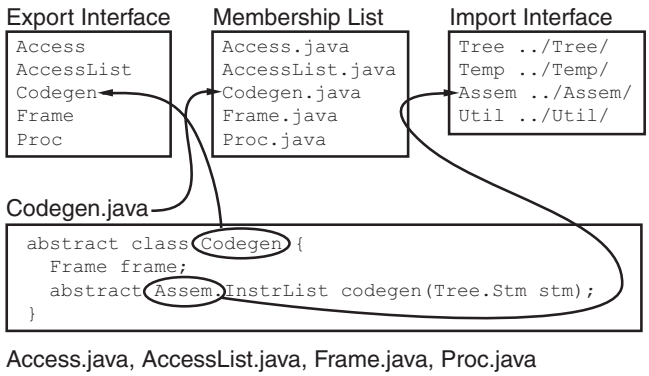

Figure 1. A fragment of the code-generation module of a compiler.

- an export interface;
- a membership list; and
- a set of import statements.

The example in Figure 1 shows what a fragment of the code-generation module of a compiler might look like.

The export interface of the module is a filter that allows only select classes to be visible externally. In this example, the class `Codegen` is listed in the export interface, which means that it can be accessed by other modules, rather than just from the source code of the current module. Any classes that are not listed in the export interface remain internal to the module, as if they were declared package-scope.

The source files that comprise the module are listed in the membership list. Every class that needs to be part of the module must be defined in one of these source files. In this case, the membership list includes the source file `Codegen.java` which defines the `Codegen` class. Explicitly keeping track of the members of a module is useful both from a software-engineering and security standpoint.

The only way to reference classes that are not in the module is through the import interface, which introduces new Java identifiers that are bound to external modules, packages[‡] or classes. In our example, the `Codegen` class needs to reference class `InstrList` from the module located in directory `../Assem/`. The import interface therefore introduces the new identifier `Assem` and binds it to the required module. All the classes that are listed in the export interface of this module can now be referenced by prefixing their names with `Assem`, e.g. `Assem.InstrList`. The names of the identifiers and directories listed in the import interface in this example match only out of convenience; no feature of the module system compels them to do so.

[‡]For compatibility with software written in Java that does not use our module system.

3. DESCRIPTION

The source files in our module system are standard Java source files, with a few exceptions:

- `package` and `import` declarations are omitted;
- the `public` and *package-scope* access modifiers in class declarations are ignored (but these access modifiers work as before for field and method declarations);
- symbols defined in the module description file may be used in the source code as identifiers;
- references to classes external to the module are allowed only via the identifiers defined in the module description file.

Java maintains separate name spaces for types, methods and variables. The name space of types in which the source files are compiled and executed is composed of all the classes that are defined within the current module and all classes imported through the module description file.

The syntax of module description files is given by:

```
Module | Library
    [ classname | classAlias ] +
is
    filename *
[ imports
    ImportStatement * ]

ImportStatement:
    moduleAlias packagename |
    moduleAlias location [ property [ , property ]*] |
    classAlias moduleAlias.classname
```

where *classname* is the simple name of a Java interface or class (e.g. `Codegen`), *filename* is the name of a Java source file[§], *location* is a relative or absolute pathname that must end with a path separator (e.g. `../Assem/`) and *moduleAlias* and *classAlias* are Java identifiers.

A typical module description file begins with the keyword `Module`. The keyword `Library` indicates that the JAR file (containing the compiled classes, the module description file and some extra information) produced by compiling the current module should include the compiled versions of all of the modules on which the current one depends. Otherwise, it would contain the compiled versions of only the classes defined by the current module.

The keyword `Module` or `Library` is followed by a list of exported symbols. Each exported symbol is a class name, either from the set of locally defined classes or one that has been imported and aliased in the imports section of the module description file.

The keyword `is` concludes the list of exported symbols and starts the enumeration of the classes that comprise the module.

[§]Another reasonable design choice would have been for the list of sources comprising a module to be made up of class files; however, since a Java source file may contain the definitions of several classes, we decided it was more convenient for it to be comprised of Java source files. The list of class files is automatically generated and stored at compile time.

The optional `imports` section can be used to establish bindings to any external classes that are to be visible to the source code of the module. Each import statement introduces a new Java identifier (*moduleAlias* or *classAlias*). A *moduleAlias* can be bound to a package or module, a *classAlias* to a particular class or interface from a module or package that has already been assigned an alias. Import statements refer to modules by their locations. A location could be a directory or a URL, although our system only currently supports directories. An import statement that binds a *moduleAlias* to a directory may optionally require that the module be annotated with one or more *properties* (see Section 5).

The aliases introduced by the module description file can be used in the source code of the module to reference classes from imported modules. The aliases may not occur bound in the source code of the module.

For backward compatibility, our module system also permits modules to import standard Java packages. The linkage specifications and security features of our module system do not apply to them; however, allowing packages to be imported smooths the migration process from standard Java.

Modules compile into JAR files, which can be digitally signed to ensure that tampering with their contents cannot go undetected.

4. FIXING JAVA PACKAGES

Our module system contains a number of features that are missing or insufficiently developed in the Java package system. The most important are explicit export interfaces and membership lists, hierarchical scalability and multiple interfaces, and convenient name-space management. These will not only be useful for software engineering, but will also enhance the security of software systems developed in Java.

4.1. Export interfaces and membership lists

A well-established principle of software engineering is that the interface of a module should be separate from its implementation. This enables a client of a module to be written and type-checked against the interface before the module's implementation is written and allows the module's implementation to be type-checked against the same interface to ensure that the implementation adheres to its own specification. Separating the interface from the implementation also aids in the construction of ADTs by making it clear which parts of the ADT form its public interface and which should remain private.

Some programming languages provide adequate support for this model of programming. C [9] allows the separation of interfaces from implementations and even the hiding of representations [10], although without enforcing it as programming discipline. Modula-3 [11] and Standard ML [12] do a good job of both separating interfaces from implementations and supporting ADTs. Java, in its native form, is lacking in both respects.

Java supports modular programming at both the class level and the package level. At the class level, the `interface` facility of the language provides support for the model of modular programming in which interfaces are separate from their implementations. It has some notable deficiencies, such as the inability to describe constructors or static methods, but classes are mainly too fine-grained a structure to be particularly suitable as units of modularity for traditional modular programming.

Java uses the package mechanism to provide support for modularity above the class level. Java packages do not separate interface from implementation—the interface is derived implicitly from `public` keywords sprinkled throughout the implementation.

Aside from the traditional software engineering goals, module systems have recently been asked to fulfill additional roles as well. With the widespread use of mobile code (e.g. applets, plugins) it has become necessary to protect systems from damage that malicious mobile code might inflict, as well as to provide environments in which mutually untrusted groups of mobile code can run simultaneously, but without danger of unwanted interaction. Since mobile applications (in Java) typically consist of several classes, it is natural that they be organized in modules. Even when this is not done explicitly, a collection of classes that comprises a mobile application is likely to share the same set of security properties and will, from the standpoint of the system within which it is running, in many respects be treated as a *de facto* module. If mobile code systems are to rely on modules to organize code, it is important that module systems assist in providing the security functionality needed for mobile code, or at the very least not interfere with the other mechanisms used to provide security.

The Java package system is unsuited for this role. The combination of implicit interfaces and the lack of explicit membership lists makes it easy for a malicious attacker to take advantage of a system for running mobile code that bases its security facilities on Java packages.

Let us consider an example. Suppose a particular application controls access to its components by declaring certain sensitive classes package-scope and only lets clients access them through public classes, which filter out any undesired uses of the private classes. This application may wish to download an untrusted, third-party plugin. The plugin may need access to the application's public fields and methods, so it is downloaded into a name space that is shared by the application and the plugin. The deficiencies of the Java package system make this insecure. An attacker could write a class that declares itself to be part of the same package as the application as part of a plugin—this is possible because Java packages do not have membership lists—and could then directly access the application's private classes and use them to malicious ends.

Our module system prevents any such security breach by using module description files which explicitly specify both the membership of a module and its public interface by listing all the classes that belong to each. Since Java packages lack a mechanism to enforce this, plugins are normally loaded by separate class loaders, so they reside in separate name spaces from the application. This effectively prevents the described attack, but at the cost of preventing the plugin from conveniently communicating with the application using the application's APIs.

There are other ways of solving security problems such as that posed by this example, for instance, by stack inspection [13]. A disadvantage of most of these schemes is that they require dynamic run-time checking and that they are needlessly restrictive. Our scheme, on the other hand, would prevent a hostile applet such as that described from even linking with the trusted application.

The module description file in Figure 2 demonstrates the use of explicit export interfaces and membership lists. Only classes defined in the listed source files are considered as part of the module. The module defines several classes, but only `Graph`, `Node` and `NodeList` are visible to clients outside the module.

Though a significant improvement from the standpoint of information hiding and program organization, the interfaces of our module system do not address the issue of separate compilation. The interfaces are merely lists of classes and do not describe their types, so an implementation cannot be type-checked against them. They present an improvement over the Java package system's implicit


```
Module
      Graph
      Node
      NodeList
is
      Graph.java
      FlowGraph.java
      Node.java
      NodeList.java
      FlowNode.java
      GraphUtils.java
```

Figure 2. The module description file of a submodule of a register allocator.

interfaces by allowing the programmer to specify the sets of classes that form a module and its public interface. We do not see a suitably non-intrusive way of adding support for separate compilation to Java, and since simplicity and convenience were important goals, we decided against extending and complicating our module system in an attempt to solve this problem.

Our approach to organizing modules is similar to, although simpler than, the mechanism for defining units in MzScheme [14], which does support separate compilation.

4.2. Hierarchical scalability and multiple interfaces

The basic ways in which our modules support information hiding are not dissimilar from those offered by Java packages. Java's module interfaces are implicit; ours are explicit, but our interface descriptions only consist of classes and do not describe public fields and methods of classes which are also part of a full interface. Though our module system is not powerful enough to fully describe the types of modules, it makes it simpler to control and enforce the visibility of member classes. The interfaces of both systems have similar access-control capabilities: a class can be either publicly visible or visible only to other classes inside the same module. The feature that sets our module system apart from Java packages, however, is the ability to structure modules so as to provide different views to different clients.

We often come across situations in which we would like a module to export a richer interface to a few select modules and a more restrictive one to everyone else. In a language like Standard ML a module can supply different export interfaces to different clients. Modula-3 also has that ability, although module interfaces in Modula-3 may not overlap in the sets of members they expose [11]. Java's methods of controlling accessibility (through making classes and their fields private, protected, package-scope, or public) are not expressive enough, so Java resorts to using a security manager to determine whether a client is allowed to access a particular restricted class at run time. The security manager suffers from a number of problems, from run-time overhead to its ability to interact only with the owner of the virtual machine and not the executing program. Its complexity and ambiguities


```
Library
      Main
is
      Main.java
      NullOutputStream.java
imports
      Codegen ../Codegen/
      RegAlloc ../RegAlloc/
      Absyn ../Absyn/
      Tree ../Tree/
      ...
      Types ../Types/
      Util ../Util/
```

Figure 3. The module description file of the top-level module of a compiler.

have made it vulnerable to security breaches and made it difficult to reason about and form security policies [15].

Suppose that there are to be two views of module M: view V_1 providing access to classes A, B, C and view V_2 providing A, D. In our module system this is accomplished by making a module M_0 containing (and exporting) A, B, C, D, a module M_1 that imports (and re-exports via aliasing) $M_0.A$, $M_0.B$, $M_0.C$ and a module M_2 that imports and re-exports $M_0.A$, $M_0.D$. There are no wrapper classes: the class $M_2.A$ is the same class as $M_1.A$.

This is an instance of hierarchical modularity, which is the idea of grouping several modules and attaching to each group its own interface. The group is itself a module whose publicly visible members can be imported by other modules. The members of the group can communicate amongst themselves through their own interfaces, which can be much less restrictive than the group's top-level interface. This approach can be applied repeatedly to create a hierarchy of modules. Hierarchical modularity was comprehensively investigated by Blume and Appel [4]; we use a similar approach for Java.

Our module system supports hierarchical modularity by allowing modules to explicitly list the sub-modules on which they depend. Modules can not only export classes that have been defined in their own source files, but also classes that have been defined in imported modules. When its module description file begins with the keyword `Library`, compiling a module produces a JAR file that includes the bytecode of all the imported modules, which are then kept hidden by the export interface. Features of the `Module` and `Library` modes of compilation are discussed further in Section 7.

Figure 3 is a module description file of the main module of a compiler; it illustrates this approach. The main module imports all the sub-modules that implement different parts of the compiler and defines only a few classes that tie the sub-modules together into a working system. One of the modules it imports is `Codegen`, the code-generation module. `Codegen` defines and exports the classes `Access`, `AccessList`, `Codegen`, `Frame` and `Proc`. Though these are visible to the source code of the top-level module, they are not publicly accessible. Only the class `Main`, the top-level interface to

the compiler, is left visible as the export interface of the group. The hierarchical structure is transparent to a user; they have no way of knowing that the compiler module is composed of sub-modules.

Apart from the need for modules to support multiple interfaces, there is another reason for introducing hierarchical modularity. Windows XP has over 38 000 000 lines of code [16]. If it were structured in just a two-level framework of classes and modules, either there would be more than 3000 modules or each module would have more than 10 000 lines. This strongly suggests that a hierarchy of modules is necessary.

4.3. Name-space management

An additional software engineering benefit is our module system's flexible and convenient name-space management scheme. Although the naming convention used with Java packages suggests that they support a hierarchical naming scheme, packages with names like `java.awt` and `java.awt.color` have no more in common than packages with completely different names.

One of the reasons for grouping code into packages is to avoid name clashes between classes. However, Java packages are themselves named, so that merely lifts the problem to the package level. Instead of a name clash between two classes called `Parser`, we might have a clash between two classes called `Util.Parser`. The accepted way of solving this problem is to give packages long, unique names. This is not a particularly appealing solution, however, since it interferes with the package system's ability to provide convenient name-space management; classes must now either be referred to individually using their cumbersome package name (e.g. `java.awt.image.renderable.RenderableImage`) or be imported *en masse* using the `*` notation, which again introduces the possibility of name clashes because the names of the imported classes are stripped of their unique package prefixes.

Our modules, on the other hand, are not named, so they do not suffer from this problem. Modules are only assigned names via the import statements of individual module description files; this type of name-space thinning makes it easy to keep their names short and simple. In source code the names of external classes are prefixed with the name of their module, so name clashes between classes with same names are easily avoided.

Our module system serves to collect together and name groups of classes. Since in our scheme modules do not have absolute or global names, another mechanism was needed to specify the modules being imported in a module description file. A convenient way to refer to these modules is by making use of the directory structure of the file system. We decided that each module would occupy a directory and a module description file would provide a way to refer to an imported module by explicitly binding the module inhabiting a particular directory to an identifier.

While this is a convenient and scalable way of building modules, it introduces into the language the idea that the relative location of a particular class (in an imported module) does not change between compile time and run time (except for classes in packages that are imported in a module declaration file; we add this feature for backward compatibility). To partially make up for the lost ability to use classes whose run-time location is unknown, our import statements can reference a module not only by its relative path name but also by a URL. We believe that in practice this gives our module system sufficient flexibility. If more is desired, however, one can imagine extending our module system to allow import statements to contain references to modules that are partially specified by external means; for example, by an environment variable. Also, our decision that a directory shall be inhabited by only

one module is for convenience: it would also be possible to refer to modules via the full path names of their module description files; this would allow multiple modules to inhabit the same directory.

The module system we present, of which the name-space management scheme is a part, is patterned upon the module system of Standard ML of New Jersey and its Compilation Manager [4]. Transplanting such a module system to Java required dealing with a number of issues that were not applicable in the SML-NJ module system. The issues unique to our work include general considerations about the interaction of a hierarchical module system with Java and Java packages, its implementation using class loaders, interaction between the security features of the module system and the Java security model and the interaction with Java reflection.

Writing secure applications in Java involves limiting the visibility of classes and preventing the run-time inspection of objects by methods such as cloning, serialization and deserialization [17]. Our module system is a significant improvement over the Java package system in addressing the first issue. The second issue is discussed in more detail in Section 7.

5. SECURE LINKING

The behavior of a program fragment depends not only on its own code, but also on the libraries with which it is linked. Under the static linking model, compiling and linking a piece of code generates an executable that is fully self-contained. The libraries with which the program is linked, as well as the finished product, are available for the programmer's perusal. He therefore has good reason to expect that the self-contained executable will behave in the desired manner, even if it is executed on a machine that has a different software environment and a different set of libraries.

Today, most executables are not fully self-contained, but need to dynamically link with system libraries when they are executed. This provides us with the flexibility to update or change parts of all programs on a system simply by swapping in a new module. Should we swap in a new I/O library to replace an old one, all executables that use that library will automatically have access to the updated code. If the executables were statically linked, on the other hand, we would have to relink each of them—inefficient and inconvenient at best.

Dynamic linking has become very popular, especially with languages such as Java, which adopt it as a key feature [18]. However, despite the proliferation of dynamic linking, only a few attempts have been made to extend the model of correctness that holds for statically linked code [19,20,21]. Programmers believe that programs will behave in their intended manner even though much of the programs' behavior depends on the system libraries of foreign and unknown systems.

This belief is mostly based on the existence of standards that seek to ensure the uniformity of library code (e.g. all JVMs and their associated system classes are expected to meet Sun's standard). There are very few guarantees, however, about adherence to a standard which are expressed in a way that *programs* can understand. The guarantees are largely implicit and informal or written in English, and cannot be reasoned about or manipulated at the level of program code. Additionally, standardization does not apply when linking with third-party libraries. The only widely used method of ensuring safe linking, and the method used by Java, is type-checking the interfaces between program fragments. Recent research has formally shown that strongly typed mobile code has desirable security properties [22] and provides ways of ensuring that type safety is preserved by the linking process [23].

Although type-checking is useful in ensuring that programs and libraries at least agree on the types they are using, it falls far short of guaranteeing that code will behave in the expected manner.

Stronger guarantees are needed, especially when a system must trust the behavior of a particular executable, such as an applet. Java often uses code signing for such purposes [7,24]. But what is the meaning of a signature on an applet? In Sun's system, from the signature of code C by key K_A we can reasonably conclude that A signed C, and nothing more. We do not know what properties A is claiming about C. However, code signing does provide a way to identify the author of a piece of code and thus attribute blame after the fact.

While providing some protection to the virtual machine against code that runs on it, code signing provides no guarantees to code about the virtual machine, nor to different code fragments about each other. Ironically, current code signing practices allow a programmer to be held responsible for the behavior of his code, while not providing him with the means of ensuring that the system on which his code is running is itself behaving in the expected manner.

We allow the programmer to require certain properties of the modules on which his code depends. If the required properties are not present, our system will not allow the program to link or execute. If they are present, the programmer can more realistically expect that his program, once linked, will behave in the desired manner. Furthermore, the programmer can annotate his own module with certain guarantees which are held to be valid once linking has succeeded. These annotations are added to a module and digitally signed after it has been compiled. We thus establish a system in which a module can assert that if the modules it imports can guarantee certain behavioral properties, then it too will behave in a certain manner.

The properties our system supports are keywords that represent statements made by an author about the behavior of his code. Our property-annotation framework does not attempt to relate the claimed properties to actual program behavior, nor does it attempt to classify properties or regulate their assignment. What we provide is a mechanism which allows statements about program behavior to be mechanically attached to modules and allows intermodule linking to be contingent upon the presence of such statements.

A programmer, for example, may want a compiler that he is writing to have the property `DoesNotPopUpAnyMisleadingDialogBoxes`. His compiler, however, uses several third-party modules, one of which is the parser module. The programmer does not have access to the source code of the parser; even if observational evidence were to suggest that the parser behaves in the desired manner, there is no guarantee that the compiler might not eventually be executed on a host where it would link with a different third-party parser which might exhibit different behavior.

The module description file of the top-level module of his compiler (Figure 4) can specify that it should link with the parser only if the parser is also annotated with the `DoesNotPopUpAnyMisleadingDialogBoxes` property. If the parsing module is not annotated with that property, the compiler will not link or execute. Now it is reasonable to annotate the top-level module's JAR file with the `DoesNotPopUpAnyMisleadingDialogBoxes` property.

Our `property` tool will take a JAR file, property name and private key. It will cryptographically hash the tuple ⟨bytecode, module description, property name⟩, sign it with the key and add this certificate to the JAR file. Thus, the JAR can accumulate certificates of the form 'key K says the module has property P'.

A hierarchical module system is integral to our scheme of attaching properties to modules. Structuring modules in dependency graphs makes it possible for a top-level module to unambiguously


```
Module
      Main
is
      Main.java
      NullOutputStream.java
imports
      ...
      Parse ../Parse/ DoesNotPopUpAnyMisleadingDialogBoxes
      ...
```

Figure 4. The module description file of the top-level module of a compiler, annotated with additional linkage directives.

declare which properties it requires of its subordinate modules in order to be able to provide certain properties of its own. A hierarchically built system also makes it much easier to reason about the properties of modules by allowing the problem to be subdivided into a number of smaller ones. Explicit module descriptions are important to this scheme because they provide a centralized framework for requiring subordinate modules to hold certain properties.

In the same way that the module system only exports the API of the topmost module and hides the interfaces of subordinate modules, it also only makes visible the properties exported at the topmost level. These may depend on the properties exported by subordinate modules, but both the properties of the subordinate modules and the way in which they affect the properties exported at the topmost level remain invisible to the clients of the top-level module.

Our property and signature system is a small step in the right direction, but we imagine that one might trust certain signers for some properties and not others. We are working on a more powerful calculus of signers and properties.

Since a program will not execute unless it is convinced that its subcomponents are usable, our approach complements traditional code signing well. Authorship can be regarded as just another property and since the author of a program can require certain properties of the modules to which his code is linked, he may now actually be willing to be held responsible for the correct behavior of his code.

6. IMPLEMENTATION

We have implemented a prototype that illustrates the features of our module system. Our prototype can be used with existing Java compilers and virtual machines.

Our modules can be translated into Java packages. Some of the features of our module system, however—in particular its ability to place various constraints on linking—cannot be expressed just using Java bytecode. Because of this, our prototype implementation needs to provide additional features both to the compiler and to the virtual machine.

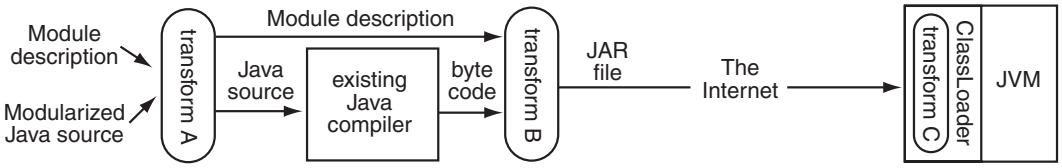

Figure 5. The implementation of our system.

6.1. Compilation

The compilation phase of our implementation is a wrapper around a standard Java compiler that consists of a preprocessing and a postprocessing step.

The job of the preprocessing phase (Figure 5, transform A) is to translate the source code used in our module system into equivalent standard Java source code. The first step of this process is to represent our modules as Java packages. Each module is assigned an artificially generated package name, mapping the hierarchical set of modules into a flat name space of packages. We rely on the assignment of artificial package names to avoid name clashes. In addition to assigning each module a package name and adding appropriate package declarations to source files, this step must also translate class references made through identifiers introduced in the module description file (henceforth called symbolic names) into class references that can be interpreted by a Java compiler (henceforth called actual names). Because identifiers in Java are classified into several name spaces, and to detect and avoid conflicts with locally bound identifiers, we have to parse the source code to determine which tokens need to be changed. As qualified names from the original source code are resolved by replacing identifiers introduced in the module description files with the package names of the modules they represent, our compilation manager ensures that the restrictions imposed by export interfaces and digital signature requirements are obeyed.

At this point our modules have been translated into ordinary Java source code and can be compiled with any standard Java compiler, without the loss of any functionality added by our module system.

The compilation phase also has a post-compilation step (Figure 5, transform B). Our modules can export symbols that have been defined in imported modules, so it is possible that several module description files need to be traversed in order to discover to which class a qualified identifier is pointing. This resolved name is the one used when the code is being compiled. Consequently, the bytecode of one module can depend on the source code of several; from a viewpoint that favors separate compilation, this is undesirable.

To allow the separate compilation of modules, we replace the resolved references in the compiled bytecode with their symbolic names. Thus all external references are again made only through identifiers defined in the module description files, releasing each compiled module from unwanted dependencies on the source code of others.

There are cases in which it is difficult to restore a resolved identifier to its original name. A particular module description file, for example, might bind two different identifiers to the same class. Preprocessing would replace the two different identifiers with the same new one. After compilation,

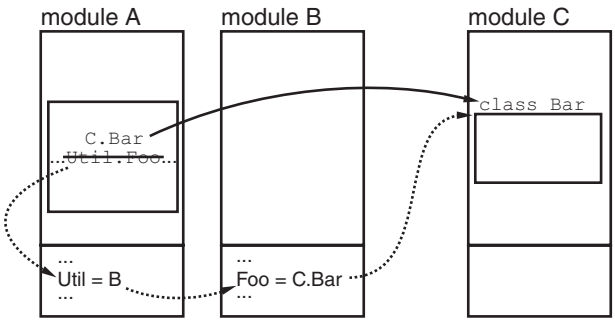

Figure 6. Resolving class references.

we might not be able to discover which of the two is which. In this situation, our compilation manager arbitrarily picks one and adds an annotation to the module's JAR file. This annotation can later be used to check whether the bindings that were used at compile time are still valid and otherwise warn that recompilation is necessary. Recompilation could be avoided by including in the module's JAR file a table that kept track of how individual symbolic names were rewritten into actual names in the module's JAR file.

Figure 6 shows an example of name rewriting. To resolve the reference to `Util.Foo`, module A first consults its module description file to discover that the identifier `Util` is bound to module B. From module B's description file we learn that the class `Foo` is re-exported rather than defined in B and that the real name of the class is `C.Bar`. The reference to `Util.Foo` is replaced by a reference to `C.Bar`. However, since module C is part of the hidden implementation of module B, it is possible that it may change after module A has been compiled. After compiling module A, therefore, the rewritten reference is returned to its original name, `Util.Foo`.

6.2. Execution in the JVM

Dynamic linking in the JVM is managed by class loaders. Class loaders were intended to be extensible to allow the virtual machine to load bytecode from sources other than the local file system. They can also be modified, however, to support arbitrary mappings from class names to objects, or even modify the bytecode of the classes they load. These features makes them useful for adding advanced language features to Java without modifying the virtual machine [8].

Each module description file sets up a mapping from identifiers to the classes they represent. The same identifier can therefore represent different classes in different modules. A request to load a certain class may also be allowed or denied depending on whether the class is signed by the digital signature required by the calling module. To deal with this issue, we have to provide the JVM with the ability to answer `loadClass` requests differently depending on the module from which they originate, which it otherwise has no way of doing.

Since `loadClass` requests are handled by the class loader that loaded the class that is making the call, our solution is to extend the `ClassLoader` class with the functionality we desire. We instantiate

a new copy of this class loader for every module that is loaded by the virtual machine. Our class loader uses the module description file to set up the appropriate class environment and control linking in the manner specified by the export filters and digital signature requirements. After the virtual machine is initialized, a wrapper class loads our customized class loader, which then loads the modules to be executed.

Each of our class loaders only has direct access to its own module description file. When a class requests that a class from a different module be fetched, the requester's class loader passes the request to the appropriate module's class loader. That class loader, in turn, verifies whether the request can be fulfilled *vis-à-vis* that module's export interface and property requirements. If the requested class is merely being re-exported, the request will be passed on to the next class loader in the chain; otherwise, the requested class will be returned.

Using class loaders to implement name-space management schemes is not a new approach [8]. Our contribution is in providing a declarative way for using them. In our system the programmer is absolved of having to manipulate class loaders himself. He merely needs to organize his code in modules and the system will ensure that class loaders are used as is appropriate to provide the required name-space separation. Because it spares the programmer from dealing with the details of a procedural approach to maintaining different name spaces, our declarative approach which deals with these details in an automated way is more reliable and easier to use.

6.3. Name hacking

The process we described for running code written using our module system is not quite complete. The ability to customize class loaders can easily be misused. If a class loader, for example, was asked to fetch the same class twice and returned two different objects, the type system would be broken and the security of the system would be compromised [19,20]. Newer JVMs have instituted stricter name-space management policies to guard against such breaches [25].

The full name of every compiled class is encoded in its bytecode. Among other restrictions, new virtual machines verify that the encoded name of a class returned in response to a `loadClass` request matches the name with which `loadClass` was invoked. Class names in our module system contain identifiers defined in module description files; these names may bear little relation to the actual package names assigned to the classes they reference. With the new security checks, it is no longer possible for our class loader to naively redirect `loadClass` requests to classes whose names do not match the requested ones.

Our solution is to rewrite the bytecode of compiled modules, replacing symbolic names (those defined through module description files) with actual ones. This is done while a class is being loaded into the virtual machine, before linking or bytecode verification (Figure 5, transform C). The procedure for resolving symbolic names is virtually identical to the one we use during preprocessing when source code is rewritten.

Since modules may re-export classes, resolving symbolic names requires tracing through module description files to locate the module in which a given class is defined. This is necessary in order to find which package name has been assigned to the module to which that class belongs. A consequence, therefore, of the bytecode rewriting is a slight restriction on the laziness of dynamic linking. A JVM might delay the loading and linking of a referenced class until the point of execution at which the class is actually needed. Our rewriting technique, on the other hand, resolves all references at load time, so at

that point it must access the module description files of all referenced modules. Since it does not need to actually load classes from the referenced modules, the chain of modules that need to be accessed for a particular reference to be resolved ends as soon as the module that defines the referenced class has been found. In the absence of re-exported classes, this will be only a single step.

It should be noted that the need to follow a chain of references all the way to the definition of the class is an artifact of our implementation, rather than a limitation of the module system. A more robust implementation of our module system would change the JVM by removing the security checks that make our method of bytecode rewriting necessary. Lazy loading of classes would then function in exactly the same way as it does in standard Java. Our approach, on the other hand, does not involve modifying the JVM itself, which makes it easily portable across different implementations.

7. THE REFLECTION API

The Java reflection API [26] is a mechanism that allows objects to be inspected and modified at run-time. It can be used to reveal information that might not be available at compile time, and lets programmers create or manipulate objects and their fields and methods even if the objects' class names, and the names of the fields and methods, are unknown until run time. Reflection is meant to be used by applications which require run-time access to objects, like debuggers, class browsers and object inspectors. It is not intended to be used when other language features would suffice; for example, as a replacement for function pointers.

Supporting Java's reflection API requires consideration of two chief issues: first, whether Java enhanced with our module system supports an implementation of reflection that provides the desired functionality and, second, whether reflection violates security by allowing access to information that the programmer is trying to protect.

Standard implementations of the reflection API do not work transparently with our module system. Our implementation of the module system requires that names of classes are changed behind the scenes in a way that is neither apparent nor intuitively understandable to the programmer. The string returned by a `getName` call would be a machine-generated class name probably meaningless to the programmer and a call to `forName` would fail since the programmer and the virtual machine would know a particular class by different names. For example, a program might contain the class `Utils.QuickSort`. During compilation, the name of the module could be translated from `Utils` to `hfrwmtlnrhwiigpejmjonobokaghkrgb` and the name of the class from `QuickSort` to `ymkctlQuickSort`, and the resulting bytecode would contain no references to `Utils` or `QuickSort`. Thus the programmer could not use `forName` to create an instance of the class `Utils.QuickSort`, because the virtual machine would have no knowledge of such a class.

It is not too difficult to provide an implementation of the reflection API that works better with our system. A first attempt to solve the problem might be using source-to-source translation to redirect method calls to a customized implementation of the reflection API. However, because of the security restrictions placed on class loaders (described in Section 6.3) and of the dynamic nature of the use of the reflection API, source-to-source translation is not quite powerful enough for our needs.

The changes we need to make are not extensive. We can modify `getName`, `forName` and other methods of the reflection API to make use of our customized class loader. Since the class loader maintains the local name space and has all the information that links class names given by the


```
Module ...
is ...
imports
      Utils ../Utils/
      Sort Utils.QuickSort
      FastSort Utils.QuickSort
```

Figure 7. Importing a class under different names. The identifiers `Sort` and `FastSort` both refer to the same class.

programmer to the machine-generated class names created during compilation, it can easily translate between one nomenclature and the other. Once the arguments of `forName` have been translated, the call can be forwarded to the standard implementation of `forName`. `getName`, if desired, can be handled so that its return value is mapped from the space of machine-generated names to the space of names given by the programmer. The latter may not always work perfectly. If several different names are used by the programmer to refer to the same class, for example, it would be difficult, given a particular usage of the machine-generated name, to tell which original name it replaces. A programmer might want to use in his code, for example, the class `Utils.QuickSort`. For experimental purposes he might want to refer to it by two names, `Sort` and `FastSort` (as shown in Figure 7). In this case the source-to-source translation that takes place before compilation would replace all the instances of the identifiers `Sort` and `FastSort` by the real name of `Utils.QuickSort`. The bytecode-to-bytecode translation would attempt to restore the original names, but since it would not know how to choose between `Sort` and `FastSort`, it would arbitrarily pick one. Any original name returned by the modified `getName` is a correct answer, however, and does not detract from the functionality of the API. Similarly to what is described in Section 6.3, keeping a persistent table of instances of symbolic names and their mappings would solve this problem.

It should be noted that, since each module has its own name space, the result of calling `getName` is meaningful only within the module in which the call is made. For example, if a module has in its description file the import statement '`Absyn ../Absyn/`', `getName` might return a class name like `Absyn.WhileExp`. In this module that answer is completely sensible, since it obviously refers to the class WhileExp which can be found in `../Absyn/`. Another module, however, might not have access to the Absyn module, or may be using the same name to refer to a completely different module; the string `Absyn.WhileExp` could thus be meaningless or it could mean something entirely different. Though it might sometimes be unexpected, this context-dependent meaning is characteristic of using class loaders to manage name spaces. A mundane example of such behavior in standard Java is that two different applets will normally be managed by different class loaders so that their name spaces do not overlap. As between modules in our module system, passing class names from one applet to the other yields uncertain information since the applets do not know how to map each other's class names to class files.

Although it is possible to implement reflection so that it works in concert with our module system, we must make sure that it does not provide a level of access to objects that violates the abstraction

principles imposed by our module system. In particular, we must make sure that reflection cannot give clients access to the module-scope classes or methods of another module.

In the simple case of non-nesting modules this does not present a problem; since modules are rewritten into packages, non-public classes and methods are protected by Java's public- and package-scope access-control mechanism. In this case each module is functionally equivalent to a Java package, so the reflection API interacts with the module as it normally would with a package in standard Java.

The situation is slightly more complicated when modules nest, because in this case we not only wish to prevent access to non-public classes of a module, but perhaps also to both the public and non-public classes of its submodules. To allow a programmer to keep submodules private, our system has two modes of compilation: in the `Module` mode, a module and its submodules are all compiled as separate packages; in the `Library` mode, the module and its submodules are collapsed, after compilation into bytecode, into a single package. The first method leaves submodules unprotected. They are compiled as ordinary modules and their public members are accessible by anyone. One might wish to use this mode to let the same compiled code be used by several clients. When using the second method, the classes from all submodules are kept in package-scope of the enclosing module; only the classes listed in the export interface of the top-level module are made public. This enables standard Java access control to prevent users of the reflection API from accessing private submodules.

One of the features of our module system is that classes can be imported and then exported under a different name. It is possible to use reflection to discover the real name of a re-exported class, whereas without reflection only the name under which it was re-exported would be known. Although this is undesirable if one wishes to completely preserve abstraction, merely knowing the real name of a class does not provide an increased level of access to it. In addition to protecting private submodules, the `Library` mode of compilation can also be used to prevent a user from learning the real names of re-exported classes.

8. CONCLUSIONS AND FUTURE WORK

Our module system is based on explicit module descriptions. Membership lists and explicit export interfaces protect module integrity. Unnamed modules and declarative import statements provide simple and convenient name-space management. Variable levels of access to modules are supported by arranging modules in hierarchies. Increased control over the linking process, implemented by allowing import statements to require modules to have specific properties, helps ensure correct program behavior in the presence of dynamic linking.

Any attempt to develop a secure programming environment is likely to be based on a module system. In the case of Java, a module system should provide modularity at the level of Java packages, but should also provide explicit interfaces, which Java packages do not. Explicit module descriptions seem to be a very useful feature, both for providing an increased level of security and for simplifying the task of designing and understanding modular software systems. Class loaders play a key role in security; our module system uses them in a principled and declarative way to enforce information hiding. We have demonstrated that the JVM is sufficiently powerful to support such an advanced module without modification.

Our module system is generally compatible with Java's notion of reflection. The standard Java reflection API does not interact well with our module system without modification. Our module system

is translated into standard Java in a straightforward enough way, however, that reimplementation of parts of the reflection API is sufficient to both restore most of the functionality of the original API and to prevent it from incorrectly allowing access to module-scope classes and methods. The current implementation places some restrictions on the lazy loading of classes. These are, however, not inherent in the module system and would not be present in a more robust implementation.

Dynamic linking is an area that deserves more study. It is important to provide guarantees—ones that programs can reason about—about the behavior of dynamically linked libraries. Only then can we trust programs that rely on them to behave in their intended manner. Our module system provides a good framework for annotating code with such guarantees. We demonstrated a method for allowing interrelated modules to require certain rudimentary properties of each other. We plan to continue work on making these linking requirements more expressive and giving modules even more control over the linking process.

REFERENCES

1. Gosling J, Joy B, Steele G, Bracha G. *The Java Language Specification* (*The Java Series*) (2nd edn). Addison-Wesley: Boston, MA, 2000.
2. Cardelli L. Program fragments, linking, and modularization. *24th ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*. ACM Press: New York, NY, 1997; 266–277.
3. Milner R, Tofte M, Harper R. *The Definition of Standard ML*. MIT Press: Boston, MA, 1990.
4. Blume M, Appel AW. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems* 1999; **21**(4):812–846.
5. Signing and checking code with Authenticode. http://msdn.microsoft.com/library/en-us/dnauth/html/signfaq.asp [1998].
6. Netscape object signing: Establishing trust for downloaded software. http://developer.netscape.com/docs/manuals/signedobj/trust/owp.htm [1997].
7. Pawlan M, Dodda S. Signed applets, browsers, and file access. *Java Developer Connection*. http://developer.java.sun.com/developer/technicalArticles/Security/Signed/index.html [April 1998].
8. Agesen O, Freund SN, Mitchell JC. Adding type parameterization to the Java language. *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*, October 1997. *SIGPLAN Notices* **32**(10). ACM Press: New York, NY, 1997.
9. Kernighan BW, Ritchie DM. *The C Programming Language* (*Prentice-Hall Software Series*) (2nd edn). Prentice-Hall: Upper Saddle River, NJ, 1988.
10. Hanson DR. *Interfaces and Implementations: Techniques for Creating Reusable Software* (*Addison-Wesley Professional Computing Series*). Addison-Wesley: Boston, MA, 1996.
11. Nelson G (ed.). *Systems Programming With Modula-3* (*Prentice-Hall Series in Innovative Technology*). Prentice-Hall: Upper Saddle River, NJ, 1991.
12. Appel AW, MacQueen DB. Separate compilation for Standard ML. *ACM Conference on Programming Language Design and Implementation*, June 1994. ACM Press: New York, NY, 1994; 13–23.
13. Wallach DS, Felten EW. Understanding Java stack inspection. *IEEE Symposium on Security and Privacy*, May 1998. IEEE Press: Los Alamitos, CA, 1998.
14. Findler RB, Flatt M. Modular object-oriented programming with units and mixins. *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, September 1998. ACM Press: New York, NY, 1998; 94–104.
15. Drew D, Felten EW, Wallach DS, Balfanz D. Java security: Web browsers and beyond. *Internet Besieged: Countering Cyberspace Scofflaws*, Denning DE, Denning PJ (eds.). ACM Press: New York, NY, 1997.
16. Appel AW. Quantitative measurements of Microsoft's source code and APIs. Plaintiff's exhibit 1722 in State of New York, *et al.* versus Microsoft Corporation. *Civil Action No. 98-1233 (CKK)*, U.S. District Court for the District of Columbia. Attachment to testimony filed April 9 2002.
17. McGraw G, Felten E. Twelve rules for developing more secure Java code. *Java World*. http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html [December 1998].
18. Lindholm T, Yellin F. *The Java Virtual Machine Specification* (2nd edn). Addison-Wesley: Boston, MA, 1999.

19. Dean D. The security of static typing with dynamic linking. *Fourth ACM Conference on Computer and Communications Security*, Zurich, Switzerland, April 1997. ACM Press: New York, NY, 1997.
20. Dean RD. Formal aspects of mobile code security. *PhD Thesis*, Princeton University: Princeton, 1999.
21. Jensen T, Le Métayer D, Thorn T. Security and dynamic class loading in Java: A formalization. *Proceedings 1998 IEEE International Conference on Computer Languages*. IEEE Press: Los Alamitos, CA, 1998.
22. Leroy X, Rouaix F. Security properties of typed applets. *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1998. ACM Press: New York, NY, 1998; 391–403.
23. Glew N, Morrisett G. Type-safe linking and modular assembly language. *Conference Record of POPL '99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1999. ACM Press: New York, NY, 1999; 250–261.
24. McGraw G, Felten E. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley and Sons: New York, NY, 1998.
25. Liang S, Bracha G. Dynamic class loading in the Java Virtual Machine. *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98). ACM SIGPLAN Notices* **33**(10):36–44. ACM Press: New York, NY, 1998.
26. Java core reflection. http://java.sun.com/products/jdk/1.2/docs/guide/reflection/spec/java-reflection.doc.html [1998].

