Object-Oriented Types for Collections of Java Classes (ooJAM)

Neil Ongkingco
Programming Tools Group
neil.ongkingco@keble.ox.ac.uk

Abstract

Module systems for handling collections of classes have been traditionally based on a container with meta-data such as versions, imports and exports. We present a novel perspective on modules by treating them as object-oriented types for class collections, and (non-formally) define relations and operations on these types. We demonstrate that these model common module operations and constraints now in use while adding the ability to express new constraints not previously possible, and show novel usage of a language features that are inspired by the type system. The type system itself is implemented as a module-enabled compiler for Java and is demonstrated on a small case study.

1. Introduction

There are module systems now, but they are lacking in elegance and functionality.

2. Module Operations

This is on the different module operations supported by current module systems.

2.1. Classpath

Jar hell. Linear sequence of the classpath limits choice of classes to load. Classloaders help, but writing a custom loader for each project is too much. OSGi helps more, but still requires a consistent class space for each classloader.

2.2. Versioning

Versions are backward compatible (usually). Versions are used as constraints for importing modules (OSGI).

2.3. Imports and Exports

Imports and exports affect accessibility and increase information hiding.

2.4. Instantiation

Different instances may be needed when name conflicts occur (e.g. different versions of the same jar)

3. Modules as Types

We now define a type system over collections of classes. We consider the module to be the type of the collection of classes that belong to it, and define relations and operations on modules.

3.1. Declaration and Membership

Modules are defined in a `.module` file that begins with a module declaration:

```
// File  org.x.y.mymodule.module
module org.x.y.mymodule;
```

Module membership is declared using syntax that was proposed by Alex Buckley for JSR 294 [3]. Membership of a compilation unit is declared by adding a module declaration. If a package declaration is present, the package name is used as the package name of the file within the module.

```
// File  C.java
module org.x.y.mymodule;
package org.x.y;
public class C {
    ...
}
```

Split packages (packages whose types belong to more than one module) have been raised as a problem by

Peter Kriens of the OSGi Alliance [6]. To avoid this, a way to specify package membership in a module allowed by declaring the module membership in the `package-info.java` file of that package.

```
// File package-info.java
module org.x.y.mymodule;
package org.x.y;
```

If the module membership specified in a compilation unit's `package-info.java` file is in conflict with that specified in the file itself, the membership declared in the file takes precedence, but a warning is issued to flag a possibly split package.

3.2. Imports

For a module to be able to access any of another module's classes, it must first import an instance of that module. This way, imports form a hard constraint on the visibility of classes.

There are two ways to import a module. The first is to import the singleton instance of the module. This is done in the following manner:

```
//imports the singleton instance
import org.x.y.othermodule;
```

Importing the singleton instance of a module allows you to share the classes of that module with other modules that also import it.

Similar to iJAM, a module may also import its **own** instance of a module. It is also allowed to rename that instance to allow for multiple instances of the same module to exist in the same context.

```
//imports an own instance
import own org.x.y.othermodule;
//imports on own instance with an alias
import own org.x.y.othermodule as myothermodule;
```

Unlike iJAM [13], renaming an instance does not implicitly allow other modules to access that instance. To allow access to other modules, the import must be exported:

```
//imports an own instance, and makes it available to
//other modules
import own org.x.y.othermodule export as othermodule;
```

The visibility of the module instances become important for module qualified name lookups and the merge and replace operations described in later sections.

3.3. Exported packages and the Module Modifier

We allow exported packages similar to `export-package` in OSGi bundles [1]. Unlike OSGi, however, exporting a package does not automatically allow other modules to gain access to these packages. A module A must first import another module B before gaining access to B's exported packages.

Exported packages are declared in the `.module` file:

```
module org.x.y.mymodule;
//allow other modules access to these packages
export package org.x.y.parser, org.x.y.lexer;
```

You may also export all the packages that belong to a module by using the * wildcard. This is the only way to export classes that belong to the default package in a module.

```
module org.x.y.mymodule;
//export all packages
export package *;
```

We also allow the **module** access modifier proposed for JSR294 [10] which applies to classes, fields and methods. Module private access only allows accesses from classes that belong to the same module.

```
module class C {
    public void publicMethod() {...};
    module void moduleMethod() {...};
}
```

3.4. Module Qualified Type References

Type references can now be qualified with module names to resolve ambiguous accesses. In this example, we have a module `myapplication` importing two instances of module `xmlparser`, which contains the class `xmlparser.Parser`. A class C in `myapplication` can then choose from which instance of `xmlparser` to load the class:

Listing 1. Module Qualified Type References

```
// File myapplication.module
module myapplication;
//import two instances of xmlparser
import own xmlparser as parser1;
import own xmlparser as parser2;

// File xmlparser.module
module xmlparser;
export package xmlparser;

```

```
// File XMLParser.java
// Classes in this file belong to module .xmlparser
module xmlparser;
package xmlparser;
public class XMLParser{...}

// File C.java
// Classes in this file belong to module .myapplication
module myapplication;
public class C{
    //This accesses the class in the first instance
    parser1 :: xmlparser.XMLParser p =
        new parser1 :: xmlparser.XMLParser();

    //This accesses the class in the second instance
    parser2 :: xmlparser.XMLParser p2 =
        new parser2 :: xmlparser.XMLParser();
}
```

Module qualified names are also supported in import declarations, and can reference exported modules from indirectly imported modules.

Listing 2. Module Qualified Imports

```
1  // file org.x.y. parserapplication .module
2  module org.x.y. parserapplication ;
3  import own parser;
4
5  // file org.x.y.parser.module
6  module org.x.y. parser ;
7  import own org.x.y.scanner export as scanner;
8  export package org.x.y. parser ;
9
10 // file org.x.y.scanner.module
11 module org.x.y.scanner;
12 export package org.x.y.scanner;
13
14 // file Parser.java
15 module parser;
16 package org.x.y. parser ;
17 public class Parser {...}
18
19 // file Scanner.java
20 module scanner;
21 package org.x.y.scanner;
22 public class Scanner {...}
23
24 // file ParserApplication .java
25 module parserapplication ;
26 package org.x.y. parserapplication ;
27
28 // single type import
29 import parser :: org.x.y. parser . Parser ;
30 //on demand import
31 import parser :: scanner :: org.x.y.scanner .*;
32
33 public class ParserApplication {
34     //uses the Parser class from module parser
35     Parser p = new Parser ();
36     //uses the Scanner class from module scanner
37     Scanner s = new Scanner();
38     ...
39 }
```

Types and packages from directly imported modules do not need to be module qualified. In the example above, one could have used

```
import org.x.y. parser . Parser ;
```

on line 29.

Types that do not belong to a module are considered to be members of the "default" module. This module has a blank name, and its singleton instance is implicitly imported by all modules. So, a module-less type `javax.swing.Action` can be accessed in this way:

Listing 3. Default Module Lookups

```
module somemodule;
import :: javax .swing.Action;
class MyClass {
    Action a = new Action();
}
```

It is not strictly necessary to always module qualify accesses to types in the default module, as long as there are no packages in the current module or its imported modules that have the same package and type name. However, if these do exist, it is necessary to use the module qualified name to access these classes to avoid an ambiguous type error.

3.5. Subtyping

We now define a subtyping relation on the module type. A module is declared to be the subtype of another module by using the **extends** keyword:

```
module org.x.y.parserV1_2
    extends org.x.y.parserV1_1;
```

A subtype module inherits all the import declarations and export package declarations of its parent module, and can add some more of its own. It also inherits the member compilation units (and the types therein) of its supertype.

If a subtype module contains a type with the identical package and type name to another type which is a member of its supertype module, this shadows the type in the supertype module for all type references that originate from the subtype module, and any other modules that import an instance of the subtype module.

The following example demonstrates module subtyping and shadowing. The module `parserv1_1` contains the type `parser.Parser`, and imports the module `scanner` which contains `scanner.Scanner`. The module `parserv1_2` then extends `parserv1_1`, while having its own version of `parser.Parser` and adding a new class `parser.XMLParser`.

Listing 4. Module Subtyping

```
// file parserV1_1.module
module parserV1_1;
import own scanner export as scanner;
//say there is a class Parser in this package
export package parser;

// file parserV1_2.module
module parserV1_2;
//say that ParserV1_2 has its own version of
//parser.Parser, and an additional class
//parser.XMLParser

// file scanner.module
module scanner;
//say there is a class Scanner in this package
export package scanner;

// file MainV11.java
module parserV1_1;
import parser.Parser;
import scanner.Scanner;
public class MainV11 {
    // this references the v1_1 parser
    Parser p = new Parser();
    Scanner s = new Scanner();
    ...
}

// file MainV12.java
module parserV1_2;
import parser.Parser;
import scanner.Scanner;
public class MainV12 {
    // this references the v1_2 parser
    Parser p = new Parser();
    // this references the inherited Scanner class
    Scanner s = new Scanner();
    ...
}

// file multiversion.module
module multiversion;
import own parserV1_1 as parserV1_1;
import own parserV1_2 as parserV1_2;

// file MultiversionMain.java
module multiversion;
public class Main {
    // this references the v1_1 parser
    parserV1_1::parser.Parser p1 =
        new parserV1_1::parser.Parser();
    // this references the v1_2 parser
    parserV1_2::parser.Parser p2 =
        new parserV1_2::parser.Parser();
}
```

3.6. Merge

It is often necessary to ensure that the exported modules of a set of modules that you import point to the same module instance, to ensure that these modules are able to share types. As an example, say you have two database backend modules `mysqlbackend` and `postgresbackend` with a method `ResultSet runQuery()`. Further suppose that `ResultSet` is defined in another module `sqltypes`, which is **own** imported by both backends. In order for the return types of both backends to be type compatible, the reference to `sqltypes` in both backends must also point to the same instance.

To ensure that the `sqltypes` instances in both `mysqlbackend` and `postgresbackend` point to the same instance, we use the **merge** operation. Merge operations take a set of merge targets, and an alias to refer to the merged module:

```
merge m1,m2,m3 [export] as alias;
```

The merge operation requires that the types of the modules being merged be the same or that one is an ancestor of the other in the subtyping relation (i.e. they are all on the same subtype path). It then creates a new instance of the most general type that is still compatible with the types of the modules being merged, and then points those module references to the new merged module. The old module instances are discarded, unless there are other references to them that are not part of the merge.

Listing 5. Merge

```
// file mysqlbackend.module
module mysqlbackend;
import own sqltypes export as sqltypes;
//say query has a class MySQLQuery
export package query;

// file postgresbackend.module
```

```
module postgresbackend;
import own sqltypes export as sqltypes ;
//say query has a class PostgresQuery
export package query;

// file sqltypes .module
module sqltypes ;
//say that sqltypes contains types . ResultSet
export package types;

// file myapplication . module
module myapplication;
import own mysqlbackend;
import own postgresbackend;
//merge the sqltypes
merge
    mysqlbackend.sqltypes ,
    postgresbackend . sqltypes
export as sqltypes ;

// file Main.java
module myapplication;
import mysqlbackend::query.MySQLQuery;
import mysqlbackend::query.PostgresQuery;
import types . ResultSet ;
public class Main {
    void someMethod(){
        ResultSet s = null;
        // this would not be possible if the sqltypes
        //module instances are different
        if (mysql) {
            s = new MySQLQuery(...).query();
        } else if ( postgres ) {
            s = new PostgresQuery (...). query ();
        }
        ...
    }
}
```

It must also be noted that the merged instance is a direct import of the module where the merge was declared and, as the example above shows, can be re-exported. This means that types in the merged module `sqltypes` can be accessed in `myapplication` without the need to use module qualifiers unless the type reference is ambiguous. As the example above shows, `ResultSet` was imported without a module qualifier in `Main.java`.

Merges are also implicitly inherited by subtype modules. The merge sequence for a module is given by the list of merge sequences of a module's supertypes, starting from the farthest ancestor. There is no way to exclude a supertype's merges, as this may reduce the module signature (the set of module references available) of the supertype module, which could lead to both internal and external clients of the module breaking.

3.7. Overrides

Subtyping allows extension or patching of a module without actually rebuilding a completely new module. However, it does have the disadvantage of being dependent on the existence of its supertype modules. To get around this limitation, we define the **overrides** relation to allow a module to completely replace another module. A module is declared to override another using the **overrides_modules** keyword:

```
module parserV2
    overrides_modules parserV1_1, parserV1_2;
```

Unlike subtyping, an overriding module does not inherit anything from the modules it overrides. However, it *must* provide the same exported modules and packages as its overridden modules to satisfy the external clients of these modules.

As is expected, override is inherited by subtype modules. This allows a subtype of an overriding module to override the same modules as its supertype.

Declaring an overriding module does not automatically change over all references to the overridden module to use the overriding module. This is done using the **replace** operation discussed below.

3.8. Replace

While merges can be used to re-bind module references, they are only applicable for modules that are subtypes of each other. As overriding modules are not necessarily subtypes of the modules they override, we introduce the **replace** operation to substitute overriding modules for overridden modules. The replace operation takes a set of target module references to rebind, and the module reference pointing to the instance to which the targets are to be bound:

```
replace m1,m2,m3 with m4;
```

The replace operation changes the binding of module references to point to an existing module instance. A module may replace another if its type is the same or a subtype of that module, or if its type overrides that of the other.

Continuing the parser example given in the subtyping section above, say we have two modules, `staticanalyzer` and `javadocgenerator`, each using `parserv1_1` and `parserv1_2` respectively. We now wish to create a new module that uses these, but updates them to `parserv2`, which overrides both `parserv1_1` and `parserv1_2`.

Listing 6. Module Replace

```
// file staticanalyzer .module
module staticanalyzer ;
import own parserv1_1 export as parser ;
...

// file javadocgenerator.module
module javadocgenerator ;
import own parserv1_2 export as parser ;
...

// file parserv2.module
module parserv2
    overrides parserv1_1 , parserv1_2 ;
...

// file myapplication.module
module myapplication;
import own staticanalyzer ;
import own javadocgenerator;
import own parserv2;

replace
    staticanalyzer :: parser ,
    javadocgenerator :: parser
    with parserv2 ;
```

Note that even though the modules `parserv1_1` and `parserv1_2` are not present in the build, it is able to compile successfully when the module references to these missing modules are replaced with the overriding module `parserv2`.

As with merges, replaces are implicitly inherited by subtype modules.

3.9. Module Interfaces

Extending the object-oriented metaphor, with module types being classes, we now define *module interfaces*, which act similarly to interfaces in Java. A module interface contains no imports, replaces, merges or even member classes, but they are allowed to have a set of export package declarations. A module implementing a module interface must contain and export the packages specified in the interface.

Listing 7. Module Interfaces

```
// file org.x.y.math.module
module org.x.y.math implements
    org.x.y. calculus ,
    org.x.y. matrices ;
export package
    org.x.y.math. calculus ,
    org.x.y.math. matrices ,
    org.x.y.math.complex;

// file org.x.y. calculus .module
module_interface org.x.y. calculus ;
export package org.x.y.math. calculus ;

// file org.x.y. matrices .module
module_interface org.x.y. matrices ;
export package org.x.y.math. matrices ;

// file engine3d.module
module engine3d;
import own org.x.y. matrices export as matrices ;

// file enginephysics .module
module enginephysics;
import own org.x.y. calculus export as calculus ;

// file myapplication .module
module myapplication;
import own engine3d;
import own enginephysics;
import own org.x.y.math;

replace
    engine3d :: matrices ,
    enginephysics :: calculus
    with org.x.y.math;
```

Rules for module interface subtyping are similar to those of interfaces in Java: interfaces can only extend interfaces, and they can not implement other interfaces.

Since interfaces do not actually contain any classes, they must be replaced by a non-interface module when compiling a fully working system. This would have to be relaxed for separate compilation of modules that reference interfaces to be possible, and this is discussed a bit more in the section on future work.

3.10. Weak Module Interfaces

It may be the case that an application developer knows the packages that he wishes to use, but the provider of the module that contains these packages did not define a module interface that specifically contains those packages. For this case, we define a *weak module interface*. Weak module interfaces act similarly to normal module interfaces, except that it is also implicitly implemented by all modules that satisfy its export package signature, even if these modules did not explicitly declare that they implemented the interface.

Continuing the example given above, weak interfaces allow us to define our own interfaces to a module without the module provider's knowledge:

Listing 8. Weak Interfaces

```
// file mymathinterface.module
weak_module_interface mymathinterface;
export package
    org.x.y.math.calculus,
    org.x.y.math.complex;

// file myfourier.module
module myfourier;
import own mymathinterface export as math;

// file myapplication.module
module myapplication;
import own myfourier;
import own org.x.y.math;
replace myfourier::math
    with org.x.y.math;
```

3.11. Type Lookup Sequence

This module system, as with any module system for Java, changes the way that type references are looked up. The following pseudocode shows how type lookup is done in the type system defined above.

Listing 9. Type Lookup

```
//lookup for unqualified names in a CU
method CU.lookup(typeName) {
  // get the module of which the CU is a member
  Module = CU.getParentModule();
  if (Module != null) {
    lookup classes in CU
    lookup classes in single type imports
    //lookup in module and its supertypes only
    Module.lookup(null, CU.package(), typeName, false)
    lookup classes in on-demand imports
    //lookup in module, including direct imports
    Module.lookup(null, CU.package(), typeName, true)
    lookup primitive types
    lookup automatic imports (java.lang)
  } else {
    normal java lookup
  }
}

//lookup for qualified names
// takes the module qualifier, package
// qualifier and typeName of a type reference
//and a boolean value lookInImports
method Module.lookup(moduleName, packageName,
            typeName, lookInImports) {
  if (special packageName (java.lang)) {
    lookup in defaultmodule
  }
  if (moduleName == null) {
    lookup in thismodule
    lookup in each successive supertype
    if (lookInImports) {
      lookup in direct imports
    }
  }
  else {
  //lookup through the module qualifier
  contextModule = lookupModule(moduleName);
  contextModule.lookup('''', packageName,
                 typeName, false);
  }
}
```

The lookup rules follow the Java way of looking up types, starting from the most local proceeding to the most global. Types are looked up first in the same compilation unit, then in the single type imports, then in the types that belong to the same package in the module and its supertype modules, then the on-demand imports, then the member types of the same package in directly imported modules, and finally to the primitive types and the implicit `java.lang` imports. It was a conscious decision to make on-demand imports come first before the lookups to directly imported modules. This is because the on-demand import is closer to the type reference being resolved, being in the same source file instead of on a separate module specification file.

3.12. Constraints to Preserve Modularity

Import own cycles are not allowed, as this leads to an infinite loop of module instance creation. However, cycles of singleton imports are allowed.

Merges cannot be done on singleton instances or modules accessed through a singleton instance. This is to make sure that other clients using the singleton instance are not affected by the change.

As with merges, replace targets can not be a singleton instance or a module accessed through a singleton instance, for similar reasons.

Merges can only be done on submodules in the same path to the root module in the submodule tree. Furthermore, this should not change the module export signature of the module being submoduled. This is to guarantee that both internal and external clients of the module can still rely on their module qualified lookups.

Exported packages in non-interface modules imply that there are classes that actually belong to that package. Otherwise module interfaces become less useful.

As mentioned, implementing a module interface means exporting the packages the interface exports. This is to ensure that the signature contract with the clients of the interface are satisfied.

All references to an interface module must be replaced. Otherwise all lookups to that reference will fail as the interface is empty.

4. Modelling Current Operations

The type system does model current operations on modules.

4.1. Classpaths

Classpaths are modeled as imports. Imports also have the advantage of allowing a choice in which class to load, and in causing an ambiguous type error at build time when more than one bundle offers a class instead of just loading (or failing to load) the (possibly) wrong class at runtime.

As mentioned previously, module-less classes are considered to be part of the "default module", and its singleton instance is implicitly imported by all modules. Completely module-less builds also work as before.

Special lookups such as the bootclasspath and java.lang classes can be modeled as being members of special modules (bootmodule and javalib), which are given preference during lookup (bootmodule first, then javalib, then everything else using the lookup rules for modules). Membership in these modules will probably be implemented in a similar way to the OSGi **extends** feature.

Split packages become less of an issue, as they can be detected at build time. Class space consistency (in OSGi) can now be violated, as long as ambiguous type references are resolved using module qualifiers.

4.2. Versioning

Subtyping and overriding model versioning. It is already done implicitly by existing module systems (newer versions are subtypes of past versions). However, the type system allows additional constraints to be specified using types. Non-backward compatibility can be modeled as changing to a completely new subtype tree:

Listing 10. Versioning Using Subtyping

```
// file appv1_1.module
module appv1_1;

// file appv1_2.module
// version 1.2 is backward compatible to 1.1
module appv1_2 extends appv1_1;

// file appv2_0.module
// but version 2 is no longer backward compatible
module appv2_0;

// file appv2_1.module
module appv2_1 extends appv2_0;
```

In this manner backward compatibility can now be expressed in and checked by the type system, instead of just being written down in separate documentation.

Overrides can be used in a similar manner. A module can be declared to only override the previous versions to which it is backward compatible.

In addition, extends allows patch releases that do not contain the entire previous version's classes, as already demonstrated in the previous section.

Interfaces can also be used to model version ranges. Versions that are in the same version range implement the same interface.

Listing 11. Interfaces as Version Ranges

```
// file appv1.module
module appv1 implements appv1to2;
 ...

// file appv2.module
module appv2 implements appv1to2, appv2to3;
 ...

// file appv3.module
module appv3 implements appv3to4, appv2to3;
 ...

// file appv4.module
module appv4 implements appv3to4;
 ...
```

These extends and implements declarations do not have to be explicit. Common usage of existing module systems mostly already assume a that a module is a subtype of another module of the same name if its version number is higher. Existing module systems can generate these types internally, while still allowing explicit declarations for patch releases using extends and using module interfaces for a less restrictive constraint on imports.

As shown on the previous section, merge and replace can be used to update references to old versions of a module with a newer version. In addition, it is now possible to make sure that modules use the exact same version of the module they import.

4.3. Imports and Exports

Imports and exports are directly implemented by modulle imports and export package. OSGi's loose coupling using export and import packages can be crudely simulated by making every module implicitly import the singleton instance of every other module, thereby making the exported packages of every module visible to any other module.

Module interfaces allow for looser module coupling, allowing for indirect references to modules depending on their export package signature. Module private allows for tighter information hiding.

4.4. Instantiation

Import own models OSGi's instances, and singleton instances mirror OSGi singletons. The module qualifier in type references and imports allow fine grained selection of the class to use.

5. Case Study: ProjectName

Show how modules can be used to create a new version without destroying the old. Also show better information hiding, and that changes to source are minimal.

6. Implications

A better solution to jar hell, no more dependence on classpath order. Also allows for fine-grained selection of which class to load.

Note that the syntax only represents the type system, it is not the only way to implement it. As mentioned, subtyping is already implicitly implemented by versions. Other existing features such as loose coupling in OSGi can also be done (i.e. by having an implicit static import of every other module inserted into every module). Also, the type system is not limited to the current implementation. For example, the import declaration can be extended to use OSGi's extensive constraint features.

Conflicts from possibly split packages can now be handled at build time instead of at class load time.

Collections of classes are now typed. What takes collections of classes as input and output? Aspects. Aspect libraries may just come closer to reality.

7. Related work

iJAM Strnisa (import own) [13]
OSGI spec v4 [1]
JSR 277 [9]
JSR 294 [10]
The IBM import from classpath dudes [4]
Component nextgen [12]
Jiazzi [8]
Scala's packages [11]
Peter Krein's stuff [6] [7]
Smart modules [2]
Various general stuff [5] [14]

8. Future work

Implement as a class loader. This will probably bring in a lot of issues related to lookup once the class is loaded. This (might) work if the JVM is modified to be aware of module qualifiers (fat chance, but who knows?).

Currently, module references are dynamically typed. Their types change if they are merged or are replaced with other modules. See if static typing of the references would make more sense, or would just be too restrictive.

Explore looser merge that synthesizes a new module if there are no conflicts (but there are problems here due to possibility of introducing class lookup ambiguity on existing modules).

Explore recursive merge that actually does merging instead of just creating a new instance. Will this break other modules?

Weaken merge/replace inheritance. Allow a super() call and the option not to use it, but guarantee that the signature of the module is not changed so as not to break the module's clients

More operations on modules than merge and replace. What about global operations? (e.g. change all references to this module/module interface to point to this new module)

8.1. Conclusions

The type system is kickass. And no, I don't want to formally define it, so don't ask.

References

[1] O. Alliance. Osgi service platform core specification. http://www.osgi.org/Specifications/HomePage, April 2007.
[2] D. Ancona, G. Lagorio, and E. Zucca. Smart modules for java-like languages. In *In 7th Intl. Workshop on Formal Techniques for Java-like Programs*, 2005.
[3] A. Buckley. Changes to the superpackage model. http://altair.cs.oswego.edu/pipermail/jsr294-modularity-eg/2008-March/000171.html.
[4] J. Corwin, D. F. Bacon, D. Grove, and C. Murthy. Mj: a rational module system for java and its applications. *SIGPLAN Not.*, 38(11):241–254, 2003.
[5] D. Hoffman. On criteria for module interfaces. *IEEE Trans. Softw. Eng.*, 16(5):537–542, 1990.

[6] P. Kriens. ijam, formalized class loading. http://www.osgi.org/blog/2007/10/ijam-formalized-class-loading.html.

[7] P. Kriens. Jsr 294 superpackages no more. http://www.osgi.org/blog/2008/04/jsr-294-superpackages-no-more.html.

[8] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proc. of OOPSLA*, Oct. 2001.

[9] S. Microsystems. Jsr 277: Java^{TM} module system. http://jcp.org/en/jsr/detail?id=277.

[10] S. Microsystems. Jsr 294: Improved modularity support in the java^{TM} programming language. http://jcp.org/en/jsr/detail?id=294.

[11] M. Odersky. The scala language specification v2.7, Jan 2004.

[12] J. Sasitorn and R. Cartwright. Component nextgen: a sound and expressive component framework for java. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 153–170, New York, NY, USA, 2007. ACM.

[13] R. Strniša. Fixing the Java Module System, in Theory and in Practice. In *FTfJP '08: 10th Workshop on Formal Techniques for Java-like Programs*, July 8, 2008.

[14] C. A. Szyperski. Import is not inheritance: Why we need both: modules and classes. In O. L. Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 19–32, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.