Language-Specific Make Technology for the Java™ Programming Language

Mikhail Dmitriev
Sun Microsystems Laboratories
UMTV29-112, 2600 Casey Avenue
Mountain View, CA 94043, USA
Mikhail.Dmitriev@sun.com

ABSTRACT

Keeping the code of a Java™ application consistent (code is consistent if all of the project classes can be recompiled together without errors) prevents late linking errors, and thus may significantly improve development turnaround time. In this paper we describe a make technology for the Java programming language, that is based on smart dependency checking, guarantees consistency of the project code, and at the same time reduces the number of source code recompilations to the minimum. After project code consistency is initially assured by complete recompilation, the information extracted from the binary classes is stored in a so-called project database. Whenever the source code for some class `C` is changed, its recompiled binary is compared to the old version of `C` preserved in the project database. As a result, we find a minimum subset of classes that depend on `C` and may be affected by the particular change made to it. These are recompiled in turn, and absence of compilation errors at this phase guarantees the consistency of the new project code. To determine which dependent classes to recompile, we categorize all source incompatible changes, and for each category establish a criterion for finding the smallest possible subset of dependent classes.

Categories and Subject Descriptors

K.6.3 [**Software Management**]: Software development; D.3.4 [**Programming Languages**]: Processors—*Incremental compilers*

General Terms

Design, Languages

Keywords

Java, dependency checking, make, development turnaround time

OOPSLA'02, November 4-8, 2002, Seattle, Washington, USA.

1. INTRODUCTION

Viewed in the beginning of its history as a language for relatively small applications, such as applets or those running on hand-held devices, the Java programming language is now used in the implementation of a wide variety of applications in just about any area. Recently there have been informal reports about some Java applications that exceed one million lines of code. Management of code volumes of this size can present a number of challenges. One of these is the time it takes to rebuild the application after changes have been made to the source code.

Though general build management may be concerned about a number of issues (e.g., think of an application written in more than one language, with parts of code generated automatically, etc.), the aspect of it that we are focussing on in this paper is minimising the number of recompilations. Once the source code has been modified, all of the modules that have changed and those that *depend* on the changed ones should be recompiled. However, unnecessary recompilations, i.e. those that would not result in changed binary files, should be avoided.

From the point of view of build management, the Java platform is distinct from most of the other industrial-strength programming languages in at least one aspect, namely its lazy linking model. It converts the final linking phase into a lazy incremental resolution process that happens every time a new class is loaded by the Java VM[1]. However, unlike e.g. Smalltalk which exploits the same linking model, the Java programming language is statically type checked. Therefore, the developers should either maintain strict *binary compatibility* (see the Java Language Specification [12], or *JLS*, as we will refer to it from now) between successive versions of the same class, or make sure that whenever an incompatible change occurs, all of the classes that depend on this change are fixed accordingly. For example, if the type of an argument of a public method `m()` of class `C` is changed from `boolean` to `int` (we are not considering method overloading), the developer should change all the calls to `C.m(boolean)` from other classes. Otherwise a `NoSuchMethodFoundException` will be thrown when execution reaches a call for `C.m(int)`.

Another feature of the Java programming language is that

[1]Some Java programming environments, e.g. Visual Cafe [19] have an option that allows the developer to compile an application into a single binary executable file. This, however, makes the application non-portable, eliminating one of the main advantages of the Java platform.

dependencies between programming modules (classes) can take many forms. Class `C` can call a method of class `D`, be `D`'s subclass, implement interface `D`, declare a local variable of type `D` in one of its methods, to name a few. Some changes to `D` will affect `C` and some not, depending on the actual kind of relation(s) between `C` and `D`. For example, consider what happens if the signature of method `D.m()` is changed. If `C` previously called this method, it, in most of the cases, should be updated and recompiled. However, if `C` simply declares a local variable of type `D`, nothing need be done with it.

Combined together and applied to large, complex applications with many internal dependencies, the above properties substantially reduce the ability of the developers to control the consistency of the resulting binary application when a change to some class is made. There is no linking of the entire application that could have diagnosed many of the problems such as "referenced member is undefined". In addition, the complex nature of dependencies between classes makes traditional hand-crafted makefiles very unreliable. Thus, developers are often left with a choice between recompiling individual changed classes (thus leaving type safety issues to be revealed at run time), or recompiling the whole application. Since the latter is time-consuming, developers tend to use mostly the former option, and only occasionally the latter. Thus the problems they experience are mostly due to broken, or, worse than that, working but now-incorrect links. This is especially true for developers who are not the original code authors and thus may not understand all of the implications of seemingly innocent changes that they make. Since linking happens lazily, a problem can manifest itself very late, significantly increasing development turnaround time, whereas using "recompile all" for large volumes of code all the time may become very annoying.

A solution that addresses this problem is called *smart dependency checking*, which can be viewed as a variant of the more general *smart recompilation* technique. The latter term was originally introduced by Tichy [18, 4] in his work that involved a Pascal compiler, though this method is generally applicable to other languages as well. In Tichy's work, the compiler's symbol table was extended to keep track of finer granularity dependencies between declarations (type definitions, constants, variables, etc.) and the items in the compilation units referencing these declarations. The possible changes to declarations were classified. For each kind of change, the dependency information was used in a test to decide whether recompilation is necessary. Later, an improved version of this technique was described in [17].

At present smart recompilation techniques are widely adopted in integrated development environments (IDE) products for popular languages. Most of the IDEs for C/C++, e.g. Microsoft Visual C++ [16] or Borland C++ Builder [6], have this feature, as well as IDEs for Pascal and its descendants, e.g. Borland Delphi [8].

Smart dependency checking was implemented in several Java IDEs and command line compilers, e.g. VisualAge for Java, Eclipse, Borland JBuilder and Jikes (see Section 5 for details). The internal design, however, is disclosed for none of them.

In this paper, we describe language-specific make technology for the Java programming language, based on smart dependency checking, and a command line tool that implements it, called *Javamake*. The important point about this technology is that it does not depend on any particular Java compiler, i.e. it obtains all the information on class dependencies from binary `.class` files. Thus, any Java compiler can be used with our tool, provided that it satisfies the minimum requirement of returning an error code which tells whether the compiler invocation was successful (all of the source files passed to it were recompiled successfully), or not (errors were detected during compilation). The tool can be used as a make utility on its own for relatively simple projects, or within a more general build procedure, (such as building an Enterprise Java application) involving many more actions than just compilation of source files.

The tool is written in Java itself and thus can be used on any platform where a Java VM is available. We believe that it can be integrated with a Java IDE in a straightforward way (for instance, this has already been done for the NetBeans™ [3] IDE). Currently the binary code of this tool is available at the Sun Microsystems Laboratories Web site, `http://www.experimentalstuff.com`.

The rest of the paper is structured as follows. Section 2 describes our technology, from the general idea to the details of its core algorithm. Section 3 presents the complete list of all *source incompatible* changes that can be made to classes. This list is used to determine dependent classes that need recompilation, based on the particular change made to a given class. In Section 4 we present some initial performance measurements. We then discuss the related work (Section 5), which, unfortunately, does not include any publications — only several software products with undisclosed internal design. Finally, we present our thoughts on future work and conclusions.

2. JAVA-SPECIFIC MAKE TECHNOLOGY

In this section, we first present the general idea of our technology. Then we discuss the most important issues of its implementation, which are the *project database* and the form of input. Finally we present the core algorithm of this technology.

2.1 The Main Idea

The main idea of our technology is to keep track of changes that the developer makes to classes that initially form a consistent set, and, depending on these changes, selectively recompile the dependent sources. Recompilation of the dependent classes may reveal errors in them and/or update the links in binary classes such that they reflect the source code correctly. Let us consider this process in more detail.

Let us call a set of classes (i.e. sources and binaries) that we want to keep consistent a *project*. We define a project to be *consistent* if all of its sources can be recompiled together without problems, and the resulting binary classes will be the same as before. If this is the case, then two important conditions hold. First, there are no dangling links between classes, such as calls to methods that do not exist or have different signatures. Second, all of the links in the binary class files correspond exactly to what the programmer would expect from looking at the application source code (an example violating this is described in Section 3).

Suppose now that the developer makes a change to one of the classes. This change may be either *source compatible* or not. Source compatibility is discussed in detail in Section 3. In short, source compatible changes are changes that do not break the contract between the changed class and other classes that reference it (its dependent classes). Changes to

private members, for example, are always source compatible, since no other top-level classes can reference such members[2]. In contrast, a source incompatible change may (though not necessarily will in each particular case) break the contract between a class and some of its dependent classes. For example, changing the signature of a public method is generally source incompatible, and would make references from other classes to this method invalid.

It turns out that the standard binary class representation (a `.class` file) contains enough information about class members, and also about other classes and members that it references, to retrieve all of the dependencies between classes. The only exceptions known to us are references to *compile-time constants* (see Section 3 for details). Thus, by comparing two binary versions of the same class, we can find out exactly what changes have been made to it. Then, by scanning the old versions of other classes, we can determine which of them depend on the given change (for example, which old classes referenced a changed method of our class). If any class has not been updated and recompiled yet, we invoke the compiler on it, possibly causing an error message. If all of the recompilations pass successfully, we know that the project is now once again in a consistent state.

As always, there are a lot of details that we must take into account as we go from this high-level description to the actual implementation. In the following sections, we will discuss the most important of them.

2.2 The Project Database

The correct working of the algorithm just presented depends on the availability of information contained in all of the old versions of project binaries, that is, the classes that comprise the previous consistent project state. If this information is unavailable for some class, we are unable to determine what changes have been made to it, and will have to recompile the entire project to ensure its consistency.

Since our technology is supposed to operate on sources and binaries within the file system, with the non-zero probability of the user deleting some binaries occasionally or recompiling them directly (i.e. not under control of our tool), the only reliable way to preserve the information contained in old binary versions is to save it separately, in a store which we call the *project database*. Apart from guaranteed preservation of old versions of binaries, there are two additional arguments justifying such a solution. First, even if we were absolutely sure that the user would never modify or delete the project binaries, we would inevitably have to save some or all of them before invoking the compiler (the number of old class versions to save would depend on how well we can predict which of them the compiler, once invoked, will change). Second, when we save the classes into our custom database, we can save only the information that is relevant for us. We can, for example, omit the method bytecodes which we do not analyse, and present the rest in a form that is fast to retrieve and easy to search. This makes the database size quite acceptable (see Section 4) and the time required to analyse the differences between the old and the new class versions very short.

[2]Nested classes (see Section 3.2) have access to private members of their enclosing classes. However, a nested class is always defined within the same source file as its enclosing class, so both classes are always compiled together. Therefore, no changes to one class can go unnoticed by the other.

The database main data structure is a hash table, which we call the *Project Class Directory* (`PCD`). The `PCD` maps each project class name to a record containing the following information about the respective class:

1. Full file system path for the `.java` file.
2. The last modification date for the `.class` file.
3. The *fingerprint* (essentially, a checksum) of the bytes in the `.class` file.
4. The reflection of a class called `ClassInfo`.

Items 1-3 above are used to determine which source and binary files have been changed. Information in item 4, obtained for the old and the new version of a changed class `C`, is used to determine what classes may be affected by the changes made to `C`. How this is done is described in detail in Section 2.4.

In addition to the above information, which is kept persistently in the database, the `PCDEntry` object in memory also contains some secondary information obtained at run time, in particular, date and fingerprint for the new class version and the full path of the class file.

The `ClassInfo` for the class contains essentially the same information as the *constant pool* (see Section 3.1), class declaration, and field and method declarations in the respective class file (see the description of the Java class file format in Chapter 4 of Java Virtual Machine Specification [15], or JVMS as we will refer to it from now). However, in contrast with the original class file format, where the data is tightly packed to minimise the total occupied space, we can afford saving this information in the form of specially grouped strings and arrays of them, which can be easily stored and retrieved, and can be searched quickly. For example, we represent the constant pool of a class as, in essence, three arrays of strings that contain, respectively, textual references to other classes, fields and methods. Thus, to find out if a class references a certain method, it is enough to compare the combination of this method's defining class, name and signature with each of the entries in the "referenced methods" array. This is certainly much faster than searching the original constant pool structure. Of course, instead of an array searched sequentially, a data structure designed for fast searching, such as a hash table, could be used. However, we feel that currently the database search is fast enough (see Section 4), and thus the current solution, which imposes lower space overhead, is quite acceptable.

2.3 Input Form

Our technology guarantees consistency for the project. Now, there is a question of how to make the Javamake tool aware of exactly which classes the user wants to include in the project.

After some consideration, we came to the conclusion that the most convenient and reliable form of input to our command line tool is a list of all of the source (`.java`) files for all of the project classes. This way, to add a class (or several classes, if in one source file there are definitions of a top-level class and a number of its nested classes) to the project, the user should simply add the source file to this list. To remove a class from the project, its source should be removed from the list. The tool currently assumes that there is one

top-level class corresponding to each source file[3], plus any number of nested classes.

The only minor problem with the above approach is that the tool has to guess about the actual full names of classes. That is because the name of the source file alone does not provide the precise information about the package name of the corresponding class. We solve this problem by making mandatory a convention suggested in the JLS, according to which the `.java` files should be placed by the programmer in directories that reflect package names of their respective classes. By enforcing this convention, we are able to determine package names for classes correctly without parsing their sources (see below for how this is done). We believe that in most cases following this strict convention should not be too inconvenient for developers.

A class file is placed by the compiler either in the same directory as its source file, or, if the `-d <destdir>` option is specified, in a subdirectory of `destdir` that reflects the class's package name. Our tool accepts the `-d` compiler option and thus knows where class files produced by the compiler should go. If `-d` is not specified, the class file location is unambiguous — it is the same directory that contains the respective `.java` file. Otherwise, the tool makes the best guess about the full class name by first assuming that the package name corresponds to the complete directory name of the `.java` file. For example, if the source file full path is `/a/b/C.java` and the destination directory is `/classes`, then we first try to find a file called `/classes/a/b/C.class`. Failing that, we try `/classes/b/C.class` and `/classes/C.class`. The tool stops on a first successful attempt and, to be on the safe side, verifies that the guessed name and the actual class name contained in the `.class` file are the same. If none of the attempts are successful, the tool issues a warning, assuming that the compiler did not manage to compile the given source successfully.

Nested classes also need attention, since, by their nature, there are no separate `.java` files for them, whereas each nested binary class is saved into a separate class file. Fortunately, each binary class file contains references to its direct nested classes in the form of a special *class file attribute* (see JVMS, §4.7.5). Thus, after parsing a class file, we can locate all of its direct nested classes and then repeat this recursively until nested classes for a given top-level class are found.

Further details of how to use the Javamake tool can be found on the web site mentioned in the introduction.

2.4 The Make Algorithm

Below is the algorithm used in our make technology.

1. The input is PJF— the set of all project source files, and the `PCD`, containing the old project classes that are mutually consistent. On the first invocation of the tool for the given project, `PCD` is empty.

2. Initialize to the empty set the following sets:
   NJF - the set of new `.java` files;
   UJF - the set of updated `.java` files;
   RJF - the set of recompiled `.java` files;
   UC - the set of updated classes;
   UCC - the set of updated classes for which version comparison has been performed;

3. For each PJF element, if the `.java` file does not have an entry in the `PCD`, add this `.java` file to NJF and UJF.

4. For each `PCD` entry, if the `.java` file recorded in it is not in PJF, remove this entry from the `PCD`.

5. For each `PCD` entry, if the `.class` file recorded in it is not found in the file system where it should be (i.e. where the compiler would place it if it is invoked now, based on this class's full name, its `.java` file full path, and the `-d` option value), add the respective `.java` file to UJF. If the `.class` file is found, but is older than its `.java` file, also add the `.java` file to UJF.

6. Pass UJF to the compiler, which will produce a new `.class` file (more than one, if there are nested classes) for each `.java` source. Add all of these `.java` files to RJF.

7. For each file in NJF, find the corresponding top-level class, plus, possibly, nested classes, as explained in Section 2.3. If some source file did not compile successfully, and thus no class files for it can be found, issue a warning. Otherwise, create a new `PCD` entry for this class and add it to the `PCD`. Remove the `.java` file from UJF.

8. For the remaining files in UJF, check if new nested classes were created for the respective top-level classes, or any of previously created nested classes do not exist anymore. Create a new `PCD` entry for each new nested class and delete the `PCD` entry for each removed nested class. Clear UJF and NJF.

9. Initialize UC as an empty set. Scan the `PCD` and put into UC each class which is not in UCC and which has been changed compared to its preserved old version. First check if the class file on the file system is newer than its old version (whose timestamp is preserved in the `PCD`). If so, compare their fingerprints to find out if the class has actually been changed. Add each changed class to UC and UCC.

10. Check each class C_i in UC, i.e. compare its old and new versions and find out if the changes are *source compatible* (see next section). For each detected source incompatible change, find all potentially affected project classes. For each such class, check if the corresponding `.java` file is in RJF. If not, add the `.java` file to UJF.

11. If UJF is not empty, go to step 6.

This algorithm never aborts, unless a real problem that looks like an internal error is detected, such as different guessed and actual names for a binary class. However, the project database is updated to a different extent depending on the results of source compilation (the tool expects that any Java compiler that it uses returns at least a general error code, telling whether all of the passed sources were compiled

[3]The Java compiler does not prevent multiple top-level classes in one source file: the only requirement is that the first declared class has the same name as the source file. However, placing more than one top-level class in a single source file is considered bad programming practice and discouraged.

successfully or not). If any of the compilations were not entirely successful, it means that the new set of sources is in an inconsistent state now (a compilation error by definition means inconsistency). Since the project database should preserve only a consistent state of the project, a radical solution would be to not update any database information related to class files until after some later invocation of our tool all of the compilations are performed successfully.

However, if a lot of sources have been updated simultaneously, and only one or two of them did not compile successfully, such an approach leads to redundant checks, on subsequent Javamake invocations, for all of the class files corresponding to the previously recompiled sources. To avoid this suboptimal behaviour, we use an observation that, on one hand, typically only a small proportion of all changed classes is modified in an incompatible way. On the other hand, a source compatible change to a class by definition never breaks the contract with any other classes, i.e. never violates project consistency. Thus, if for some class the old and the new versions are compared and no source incompatible changes are found, we can always safely update this class's representation in the database. However, we still update the information for classes changed in an incompatible way only if no compilation errors at all were detected.

3. SOURCE INCOMPATIBLE CHANGES

A source incompatible change to a class is a change to its `.java` source which, once made, may break the contract between this class and its dependent classes. This happens if either:

- successful joint compilation of the sources for this class and its dependent classes becomes impossible; or
- an application that includes this class and its dependent classes runs incorrectly, unless the dependent classes are recompiled.

An example of a source incompatible change which causes the first of the above problems is a change to a signature of a public method in class `C`. In the general case, another class which calls this method and is not updated accordingly, will not pass compilation with the new version of `C`. An example of the second kind of a source incompatible change is adding a `static` method `m()`to class `C`, which overloads a method with the same name and signature in `C`'s superclass `S`. The problem here is as follows: if there are any calls of the form `C.m()` in other sources, some compilers could have compiled them into hard-wired calls to `S.m()`. Therefore, even in presence of the actual `C.m()` method, these classes will still call `S.m()`, until they are recompiled.

We have formulated the above definition of source compatibility after studying Chapter 13 of the JLS, which is the only place in this book where source compatibility is mentioned, though never detailed. JLS in fact describes only *binary compatibility* issues. Quoting the JLS, "a change to a type is binary compatible with (equivalently, does not break binary compatibility with) pre-existing binaries if pre-existing binaries that previously linked without error will continue to link without error". So, binary compatibility is concerned about successful class linking and does not always guarantee formally correct program execution, i.e. the execution that the programmer would expect looking at the source code of classes (for examples of such a behaviour see e.g. JLS §13.4.23). Source compatibility, on the other hand, is about successful class compilation and correct program execution. Several examples in the JLS show that the requirement for source compatibility is stronger than the requirement for binary compatibility, i.e. every source compatible change is also binary compatible, but not every binary compatible change is source compatible.

Our choice to maintain source, rather than binary compatibility, was made to ensure that the existing programs work precisely as expected after a change, and thus the application developers are given the most reliable support.

Tables 1 – 5 define all of the source incompatible changes to classes which are detected by Javamake, and for each such change the classes that may be affected by it. The list of source incompatible changes was created by studying Chapter 13 of JLS, and also the work by Drossopoulou *et al.* [11] (the latter group worked on issues related to formal verification of Java programs and has discovered some "holes" in the JLS). Then we determined what constructs in other classes, if not changed, will cause source incompatibility problems. These statements refer to the class or interface being changed (denoted `C` or `I`) in one way or another. After compilation, these references are placed in various parts of the binary Java class (see details in the next section). Summarising the above information, we can finally derive criteria for determining a set of classes that are affected by a particular change. In some cases, however, we extended these sets of classes, to avoid implementing complex procedures that parse method bytecodes. For example, it is clear that if class `C` is made `abstract`, then only those classes which previously contained the "`new C()`" expression, need to be recompiled. We, however, replace the test for the presence of a bytecode corresponding to "`new C()`" with a simpler test for a reference to class `C` from the *constant pool* (see next section) of another class. Our observations show that usually this kind of change is rather rare. When it happens, the number of classes recompiled unnecessarily and, what is more important, the overall time overhead caused by such a lack of precision, is usually very tolerable, i.e. is of order of 10-20 per cent of the total execution time.

In these tables, we denote a changed class or interface `p.C` or `p.I`, where `p` is a full package name and `C` is the remainder of the class name (for top-level classes it is a simple class name, and for nested classes it is the combination of all of its enclosing class names plus its simple name).

Several different forms of the "class `A` *references* class `B`" expression are used throughout these tables. How binary Java classes can be referenced from other classes, and how we define several qualified kinds of such references (for example "class `A` references class `B` *directly from the constant pool*" or "*as a thrown exception*") is explained in the next section. As for class members, i.e. fields and methods, they can be referenced from classes only through the constant pool, so for the sake of brevity we simply say that "class `A` references member `m`" everywhere.

Whenever in the "Potentially affected classes" part it is not specified in which packages, subclasses of other classes, etc. we should look up affected classes, it is implied that the search is restricted to the applicable subset of classes determined by the accessibility of the class member under consideration. For example, if a `protected` field of class `p.C` is deleted, it makes sense to look for references to this field only in classes in the same package `p` or subclasses of `p.C`.

Table 1: Changes to class and interface modifiers and affected classes.

Class and Interface Modifiers	
Change to class `p.C` *or interface* `p.I`	*Potentially affected classes*
Adding `abstract` modifier	Classes referencing `p.C` (but not `p.C` array class) *directly* (see Section 3.1) from their constant pools.
Adding `final` modifier	Immediate subclasses of `p.C`.
`public` class made `protected`	Each class that is not a member of package `p` and not a direct or indirect subclass of `C`'s directly enclosing class, that references[5] `p.C`.
`public` class made default-access	Each class that is not a member of package `p`, that references `p.C`.
`protected` class made default-access	Each class that is not a member of package `p`, whose direct or indirect superclass is `C`'s directly enclosing class, that references `p.C`.
`public` class made `private`	Each class whose top level enclosing class is different from that of `C`, that references `p.C`.
`protected` class made `private`	Each class whose direct or indirect superclass is `C`'s directly enclosing class, or which is a member of package `p`, whose top level enclosing class is different from that of `C`, that references `p.C`.
Default-access class made `private`	Each class that is a member of package `p`, whose top level enclosing class is different from that of `C`, that references `p.C`.

Table 2: Changes to superclass/superinterface declarations and affected classes.

Superclasses and Superinterfaces	
Change to class `p.C` *or interface* `p.I`	*Potentially affected classes*
Deleting class (interface) `S` from the list of superclasses (directly or indirectly implemented interfaces) of class `p.C`	Each class that references **both** `S` and `p.C` (or array classes for one or both) as follows: from their constant pools (directly or *indirectly*), as a type of a data field, as a type in a method (constructor) signature or a thrown exception, as a direct or indirect superclass or superinterface.

Table 3: Changes to field declarations and affected classes.

Class and Interface Field Declarations	
Change to class `p.C` *or interface* `p.I`	*Potentially affected classes*
Adding a non-private field `f` to `p.C` that hides a non-private field with the same name in `p.C`'s superclass `S`[6]	Each class that references `C.f`
Deleting a non-private field `f` from `p.C`[7]	Each class that references `p.C.f`
`public` field `f` made `protected`	Each class that is not a member of package `p`, that references `p.C.f`[8].
`public` field `f` made default-access	Each class that is not a member of package `p`, that referencies `p.C.f`.
`protected` field `f` made default-access	Each class that is not a member of package `p` and whose direct or indirect superclass is `p.C`, that references `p.C.f`.
`public` field `f` made `private`	Each class that references `p.C.f`
`protected` field `f` made `private`	Each class that is a member of package `p` or whose direct or indirect superclass is `p.C`, that references `p.C.f`.
default-access field `f` made `private`	Each class that is a member of package `p` that references `p.C.f`.
Non-private field `f` made `final`	Each class that reference `p.C.f`
A `final` modifier is deleted for a non-private `f` field which is a *compile-time constant*, or its initial value changed, or such a field is deleted[9]	Each class from which `p.C.f` is accessible.
Non-private instance field `f` made `static` or vice versa	Each class that references `p.C.f`.
Non-private field `f` made `volatile` or vice versa	Each class that references `p.C.f`.

Table 4: Changes to interface method declarations and affected classes.

Method Declarations in Interfaces	
Change to interface `p.I`	*Potentially affected classes*
Adding method `m()`[10]	Each non-abstract class that directly implements `p.I` or some subtype of `p.I`, or that is a direct subclass of an abstract class that implements `p.I` directly or indirectly.
Deleting method `m()`	Each class that references `I.m()`

Table 5: Changes to class method/constructor declarations and affected classes.

Method and Constructor Declarations in Classes	
Change to class `p.C`	*Potentially affected classes*
Deleting non-private method `m()` or constructor `C()`	Each class that references `p.C.m()` or `p.C()`
Adding one or more constructors, all of which have non-zero number of parameters, to class `p.C`, which previously had no explicitly declared constructors[11]	Each class that references parameterless constructor `C()`
`public` method `m()` or constructor `C()` made `protected`	Each class that is not a member of package `p`, that references `p.C.m()` or `p.C()`.
`public` method `m()` or constructor `C()` made default-access	Each class that is not a member of package `p`, that references `p.C.m()` or `p.C()`.
`protected` method `m()` or constructor `C()` made default-access	Each class that is not a member of package `p` and whose direct or indirect superclass is `p.C`, that references `p.C.m()` or `p.C()`.
`public` method `m()` or constructor `C()` made `private`	Each class that references `p.C.m()` or `p.C()`.
`protected` method `m()` or constructor `C()` made `private`	Each class that is a member of package `p` or whose direct or indirect superclass is `p.C`, that references `p.C.m()` or `p.C()`.
Default-access method `m()` or constructor `C()` made `private`	Each class that is a member of package `p`, that references `p.C.m()` or `p.C()`.
Changing the signature (includes the result type) of a non-private method `m()`	Each class that references `p.C.m()`
Making a non-private method `m()` `abstract`	Each class that references `p.C.m()`
Making a non-private method `m()` `final`[12]	Each class that is a direct or indirect subclass of `p.C` and implements `m()`
Making a non-private instance method `m()` `static` or vice versa	Each class that references `p.C.m()`.
Extending the set of exceptions thrown by a non-private method `m()` or constructor `C()`	Each class that references `p.C.m()` or `p.C()`.
Adding to `p.C` a non-private method `m(xxx)` or constructor `C(xxx)` which overloads an existing (declared or inherited) method `m(yyy)` or constructor `C(yyy)`[13]	Each class that references `p.C.m(yyy)` or `p.C(yyy)`.
Adding a non-private `static` method `m()` to `p.C`, that hides an inherited static method with the same name and signature defined in class `Csuper`[14]	Each class that references `Csuper.m()`.
Adding a non-private method `m()` to `p.C`, when a method with the same name is declared in `C`'s subclass `D`, such that now `m()` in `D` overrides or overloads `m()` in `p.C` [15]	`D` and each class that references `D.m()`.
Making method `m()` more accessible	Each class that is a direct or indirect subclass of `p.C`, and defines `m()` with a less accessible set of modifiers.
Adding `abstract` method `m()` to `p.C`	Each concrete direct or indirect subclass of `p.C` that does not define or inherit a concrete implementation of `m()`.
Deleting a concrete method `m()`, which is declared `abstract` in `Csuper`, from abstract class `p.C`	Each concrete direct or indirect subclass of `p.C` that does not define or inherit a different concrete implementation of `m()`.
Changing the kind of an exception class `p.C` from unchecked to checked	Each class that references `p.C` directly from the constant pool, and each class that references any method that throws `p.C`.

An unexpected conclusion that one can make looking at these tables is that *most* of the kinds of changes to Java classes are formally source incompatible, and under certain circumstances can break something in a particular project (in practice, though, this does not happen very often, since at least humans tend to make compatible changes much more often than incompatible ones). Another conclusion is that such properties of the Java language as nested classes, abstract and interface types, and various accessibility modifiers, result in most of the non-trivial content of these tables, i.e. entries saying more than just "all classes referencing the changed member". To some surprise, the "adding a method to an interface" change has resulted in the most complex criteria for determining affected classes.

[5]From now and until the end of this table, "references" means "references `p.C` (or its array class) as follows: directly from the constant pool, as a type of a declared data field, as a type in a method (constructor) signature or a thrown exception, as a directly implemented interface or a direct superclass".

[6]This is a binary compatible change (JLS, §13.4.7). However, it is source incompatible.

[7]This means that the field is no longer *declared* in class `C`. According to the Java class bytecode specification, all references to a field in other classes have a strict form `C.f`, so even if `f` is moved to a superclass of `C`, classes compiled against the old version of `C` will not run with the new version.

[8]One may think that a more appropriate criterion could be "Each class that is not a member of package `p` *and is not* `C`*'s direct or indirect subclass*, that references `p.C.f`". However, according to the JLS §6.6.2, subclasses of `p.C` that are not members of package `p` have to follow additional restrictions to access protected members of `p.C`, so their code needs checking as well. The same comment applies to `protected` methods.

[9]A field is called compile-time constant if it is `final`, `static` and initialised with a compile-time constant expression. The above change is binary compatible (JLS, §13.4.8). However, it is source incompatible, since pre-existing Java binaries include the *value* of the constant, rather than a reference to it, and will not see any new value for the constant, unless they are recompiled. This is a side-effect of the support of conditional compilation in Java, as discussed in JLS §14.19.

[10]This is a binary compatible change (JLS, §13.5.3). However, it is source incompatible — for example, some class implementing `I` may *already* define `m()`, but non-public.

[11]This is equivalent to deletion of the only existing parameterless constructor.

[12]Making a static method final is a binary compatible change (JLS, §13.4.16). However, it is source incompatible.

[13]This is a binary compatible change (JLS, §13.4.22). However, it is generally source incompatible, because for some dependent classes a problem of finding the most specific (JLS, §15.11.2.2) method or constructor can arise.

[14]This is a binary compatible change (JLS, §13.4.23). However, it is source incompatible, since due to possible strict references to method `Csuper.m()` from bytecodes of `C`'s dependent classes, their behaviour will be not as expected, if their sources are not recompiled.

[15]Citing the JLS, §13.4.5, "Adding a method that has the same name, accessibility, signature, and return type as a method in a superclass or a subclass is a binary compatible change". However, in a more general case, i.e. when the accessibility or the signature of the added method may be different, this kind of change is generally source incompatible. The compilation errors like "method made less accessible in a subclass" or "failure to find most specific method" may arise, or method calls can be re-bound to different methods during recompilation.

3.1 References in Java Class Files

Our tables of incompatible changes present the conditions under which classes should be recompiled in a technically precise form. Therefore, we use some special terms, first introduced in the specification of binary class format in JVMS, plus our own terms *direct* and *indirect references* from one binary Java class to another. Below is a short summary of class referencing issues.

A binary Java class references other classes and their members only symbolically, i.e. by textual names. All these names are stored in the form of string constants in this class's *constant pool*, which is a table of variable-length structures representing various constants that are referred to by the class. How can we find out which string constants contain real class names? We can do that firstly by tracing references from other entries of the constant pool, which have more explicit types, denoted as follows: `CONSTANT_Class`, `CONSTANT_Fieldref`, `CONSTANT_Methodref` and `CONSTANT_InterfaceMethodref`. These entries are added to the constant pool of class `C` if `C` references some other class `D` explicitly, or references `D`'s fields or methods. In addition, every class that declares its own fields or methods has a non-empty *Field table* and *Method table*. Entries of this table, denoted `field_info` and `method_info`, contain indexes into the constant pool, and at these indexes there are string constants (constant pool entries of type `CONSTANT_Utf8`) representing the names and signatures of the respective fields and methods. The structure of each of the presented constant pool entry types, as well as of a Field/Method table entry, is depicted in Figure 1. Arrows represent indexes of other entries which the given entry references.

By saying that class `C` is referenced *directly* from the constant pool of another class, say `D`, we mean just that there is a separate `CONSTANT_Class` entry for class `C` in `D`'s constant pool. This kind of reference appears, for example, if a "`new C()`" statement or a declaration of a local variable of type `C` appears somewhere in the source code of `D`. `CONSTANT_Class` entries for `C` also appear in `D`'s constant pool (in addition to the relevant `CONSTANT_Fieldref` or `CONSTANT_Methodref` entries), if class `D` uses a field or calls a method declared in class `C`.

On the other hand, if, for example, class `D` calls a "`void m(C c)`" method of yet another class `E`, then the only reference from `D` to `C` will be through the signature of method `m(C c)`, which is stored in `D`'s constant pool. We call this an *indirect* reference to `C` from the constant pool of `D`.

One additional complexity that we have to cope with is due to a change in the Second Edition of the JLS, which, however, has only recently been reflected in the Sun's `javac` compiler (we are presently unaware of any other Java compilers changed in the same way). Consider the following example:

```
class A { public void m() { }; }
class B extends A { }

class C {
  public static void main(String args[]) {
    A a = new A(); a.m();
    B b = new B(); b.m();
  }
}
```

In binary classes created by `javac` prior to JDK™ ver-

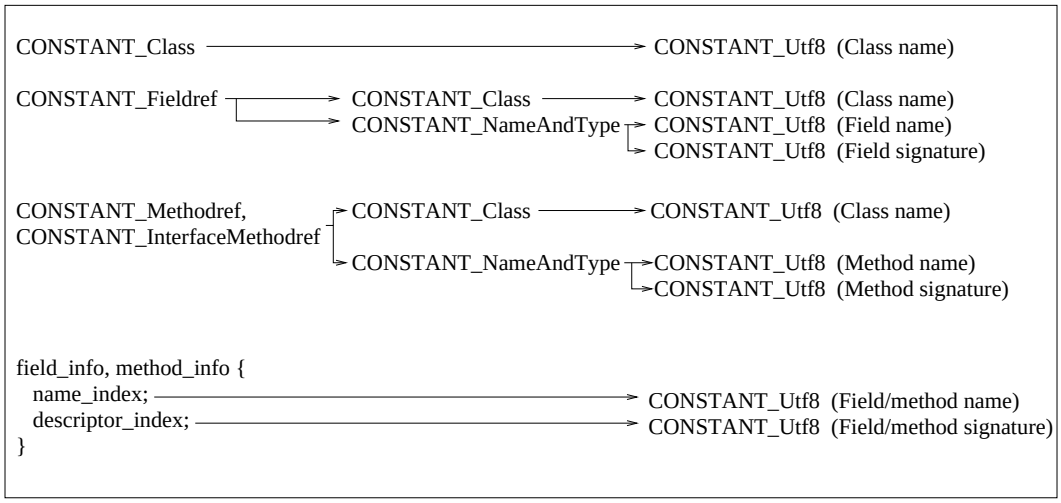

Figure 1: Binary Class Constant Pool Entries Structure

sion 1.4, both of the calls to `m()` in the above example would result in the same reference to method `A.m()` in `C`'s constant pool. However, since version 1.4, `javac` creates a separate, receiver type dependent, member reference in cases such as the `b.m()` call above. This call will now be compiled into the reference to (non-existent in this case) method `B.m()`. At run time, the JVM resolves such a reference correctly, depending on the real structure of classes `A` and `B`. This change to the JLS was made in order to increase the robustness of applications assembled from classes coming from independent sources. However, to handle such, effectively "fake", references in Javamake, we have to take special measures. If we detect a change to member `m` of class `A`, we scan project classes that may depend on `A`, looking for references to member `m`. Whenever we find such a reference, we analyse its class part, checking whether the referenced class is `A` or a subclass of `A`, say `B`. In the latter case we check if `m` is overridden in `B`, or in any class between `A` and `B` in the hierarchy. If not, then the reference is really to `A.m` and should be treated accordingly.

3.2 About Nested Classes

Nested classes are classes defined within bodies of other classes. They were first introduced in JDK 1.1 to support the new model of event handling in GUI (Graphical User Interface) programs. In such programs, nested *anonymous* classes are used essentially as placeholders for methods that handle events. An instance of a nested class which defines method `m()` is first passed to a GUI library (AWT or Swing) class, and then `m()` is called when an appropriate event is generated. In effect, an instance of an anonymous class is used instead of what would be a function pointer in some other language. The advantage of using nested classes instead of top-level classes in this particular case is that we can define a number of classes with event handling methods (the latter are typically very short) "in place", within the body of a single top-level class. Otherwise this model of event handling would require us to create countless number of separate classes, each in a separate source file, and each responsible for handling, e.g., a single menu item.

Static nested classes is another useful category of nested classes. These are in all respects similar to top-level classes, but also have access to private members defined in their enclosing class. It is a good idea to make some class a static nested one if it is small and probably too insignificant to put in a separate source file, yet the developer would like to avoid the bad practice of putting multiple top-level classes in one source file.

Nested classes introduce additional complexity that our tool has to deal with. First of all, they do not have their own source files, thus we must take additional steps to locate such classes and keep track of them (see Section 2.3). In addition, it turns out that not all kinds of nested classes should be treated in the same way as top-level classes — namely, it does not make sense to keep track of versions for some of them. To see why, and to better understand the issues related to nested classes, let us first present a general picture of nested classes in the Java programming language.

The description of the various kinds of nested classes is scattered about the Java Language Specification, Second Edition [12] in a number of sections in different chapters. Instead of reproducing their contents in this paper, we present a diagram (see Figure 2), on which the taxonomy of nested classes is depicted, with a reference to the appropriate section of the JLS for each item.

Any non-member nested class, which includes local and anonymous classes, may not be referenced by the source code of any class defined outside the immediately enclosing source code block for this class. Thus, no change made to a non-member class `C`, even its complete deletion, can affect any other class except those defined within the same source code block. These latter classes will be recompiled anyway when `C` changes, since they reside in the same source file as `C`. Therefore, versions of non-member nested classes need not be compared during step 10 of the Javamake algorithm (see Section 2.4). Instead, the new versions are saved unconditionally in the project database whenever such a class is changed. We still have to keep track of non-member nested classes during other phases of Javamake execution since such classes themselves may reference other (top-level and member) classes, and thus may need recompilation in case of incompatible changes to the latter.

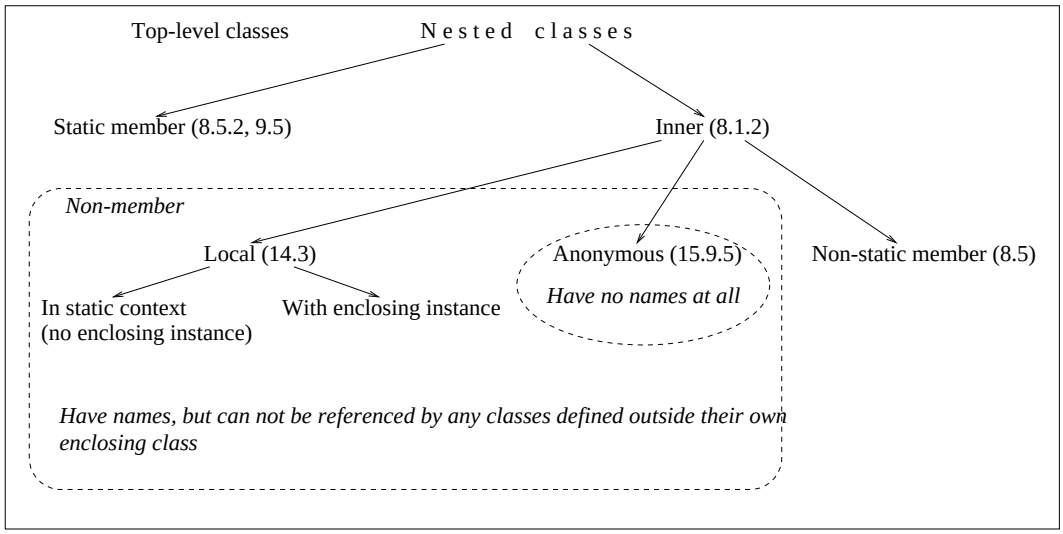

Figure 2: Taxonomy of nested classes in the Java programming language

4. EVALUATION

In this section, we first present the results of performance measurements for Javamake. We then compare its "quality", measured as the number of correctly spotted inconsistent changes, with that for several other tools supporting smart dependency checking.

4.1 Performance

It is essential for a make tool like Javamake to keep the overhead above the time it takes to recompile the changed sources to the minimum, since, after all, such a tool is designed to minimise development turnaround time. However, certain kinds of overhead are inherent to all make tools that work with files, e.g. they all have to read time stamps of all of the project files, which is a relatively time-consuming operation. In our case it also makes sense to compare the time it takes to run our tool with the time it takes to recompile the whole project, since in the absence of smart dependency checking technology the latter is the only way to ensure that the project is consistent.

Our experimental setup consisted of a Sun Enterprise™ 3500 machine with four UltraSparc™ II processors, with 4 GB RAM, running the Solaris™ Operating Environment, version 2.8. The HotSpot™ Java VM and the `javac` Java compiler from JDK Version 1.4 were used. As compilation targets we used two projects, *jEdit* and *Netbeans*, freely available on the Internet (at Web sites `www.sourceforge.net` and `www.netbeans.org` respectively). From the latter project, we compiled two code subsets: the "core" and the "extended" one, differing by size. We refer to these subsets as NBc and NBe in the following discussion. The characteristics of these projects are presented in Table 6.

The results of the first set of our measurements are presented in Table 7. We compared the time it takes Javamake and the popular make tool GNU make (version 3.78, available from `www.gnu.org`) to run for the same project, and the time it takes to recompile the whole project from scratch. For each tool, we measured three values: the time it takes to run it over the completely unchanged set of sources, the time it takes to run the tool plus the compiler when an incompatible change is made to a single source file (though this particular change does not break anything in the project), and the time when 5 incompatible changes are made to 5 source files (again, nothing is broken). The incompatible changes in each case were: making a public class package-private, making a public field package-private, making a public method package-private, making a method final, and making protected method public. In the zero changes case, we observe the overhead due to functionality which is common to Javamake and GNU make (time stamp checking for files). Comparing the other two execution times for GNU make and Javamake indicates additional overhead due to dependency checking in Javamake, and can be viewed as a typical "response time" of this tool that one is likely to expect in practice.

Table 6: Target projects characteristics

Project	KLOC	Number of sources	Number of classes
JEdit	72.2	246	466
Netbeans "core" (NBc)	230	843	2373
Netbeans "extended" (NBe)	611	2580	6558

The measurements show that Javamake runs slower than GNU make generally by a factor of 1.2...1.4, depending on the project characteristics. This is an expected, even a very good result, given that Javamake runs on top of a JVM[16] and performs a number of operations, such as reading a project database from disk, that the traditional make utility does not perform. We believe that the absolute overhead of Javamake is also within acceptable bounds in all cases.

[16]For small projects and small number of changes, the constant JVM startup time becomes a considerable part of the total Javamake execution time — hence a notable relative performance degradation of Javamake recorded in the first line of Table 7.

Table 7: Javamake vs. GNU make performance evaluation

Project	No of changes	GNU make time, sec	Javamake time, sec	Recompile all time, sec
JEdit	0	1.27 (1.00)	2.12 (1.67)	19.95
	1	2.48 (1.00)	3.29 (1.33)	
	5	3.66 (1.00)	4.50 (1.23)	
NBc	0	4.49 (1.00)	6.41 (1.43)	70.54
	1	7.47 (1.00)	9.43 (1.26)	
	5	8.74 (1.00)	10.45 (1.20)	
NBe	0	13.95 (1.00)	18.25 (1.31)	265.09
	1	16.72 (1.00)	22.57 (1.35)	
	5	17.98 (1.00)	23.69 (1.31)	

Table 8: Project database characteristics

Project	JEdit	NBc	NBe
PDB size, KB	404	1670	4011
PDB create time, sec	1.60 (41%)	7.81 (78%)	24.52 (106%)
PDB read time, sec	0.34 (9%)	1.05 (11%)	1.92 (8%)
PDB write time, sec	0.37 (10%)	1.07 (11%)	2.08 (9%)

That is, developers would be willing to pay this small cost for safety that they get — especially if it is compared to the much greater (by a factor of 5 to 10) time needed for complete recompilation of the whole project.

The data on the disk space and performance overhead due to the presence of a project database is presented in Table 8. We have measured the size of the project database and several performance characteristics. The first one is the initial time to create a project database on the first invocation of the tool for the given project. This time is spent in parsing `.class` files and creating the corresponding internal data structures, and is a one-time overhead, since on subsequent invocations usually only a fraction of classes changes, and the time to parse them is negligible. However, to read and write existing internal data structures to/from disk also requires time, and we present the corresponding measurements, both absolute and in percentage to the average total Javamake execution time for each project.

We have observed that project database size grows practically linearly with the number of source code lines. The ratio between the above two values seems to vary little for all of the projects on which we tested our tool, and is between 6.5 and 7.5 kbytes per 1 KLOC (thousand lines of source code). This yields e.g. a 4MB database for a project consisting of 611,000 lines of code, which we believe is quite modest overhead. For comparison, sources for this project occupy about 25MB of disk space, classes — about 10MB, and the `javac` compiler should be invoked with the JVM heap size set to about 128MB to make it recompile all these sources in one go.

Table 9: Total dependency checking time, ms

Number of changes	JEdit	NBc	NBe
1	13 (0.40%)	21 (0.22%)	69 (0.31%)
5	49 (1.08%)	78 (0.74%)	211 (0.89%)

The time to create a project database can be quite significant, but since the project database is created once for a project, and after that a developer needs to re-create it rarely if at all, there is no practical reason to take this overhead into serious consideration. The time to read and write a project database appears to be pretty stable, each taking about 10 per cent of the average total execution time of the tool.

Dependency checking is the main part of our technology, so it was important to determine how much time it consumes. The results of the measurements, presented in Table 9, were a pleasant surprise for us (note that results in this table are in milliseconds, unlike in the other tables). It appears that dependency checking, which may involve traversing the whole project database many times, nevertheless happens instantly compared to other operations. Its duration can vary depending on the project and on the particular change, but in none of our tests, and in further informal evaluation of our tool, did the proportion of time spent in dependency checking exceed 2 per cent of the total time. Thus it turns out that most of the time that our tool runs is spent in project database reading/writing, file timestamp checking, and source file recompilation. In Section 6 we explain that in certain circumstances the first two sources of overhead can be eliminated almost completely.

4.2 Detecting Incompatible Changes — How Well Other Systems Do

A key characteristic of the quality of a dependency checking tool (provided that its performance is at least satisfactory) is its ability to correctly detect all possible incompatible changes and report inconsistencies caused by any such change. In order to determine how good in this respect our tool is, compared to others, we evaluated three presently available tools that support smart dependency checking: Borland JBuilder version 6.0, Eclipse version 2.5 and Jikes version 1.15 (detailed information on all these tools is presented in Section 5).

To compare the capability of these tools for detecting and handling incompatible changes, we created a set of test cases according to the tables given in Section 3. Each of these tests is a small set of Java classes that is initially consistent. Then an incompatible change is made, and the tool's "smart make" command is invoked. We would then see if the resulting inconsistencies in dependent code were correctly detected, or, in case of a change which is binary compatible (though source incompatible), if dependent code was recompiled properly.

Our evaluation has shown that of the three above tools, one tool could detect all of the incompatible changes as efficiently as Javamake. One other tool was unable to detect one kind of change. As a result of this undetected change, the recompiled application would crash due to an unsatisfied link error. Finally, the third tool was unable to detect

eight kinds of source incompatible changes. However, all these changes are binary compatible, so the resulting application would not crash. It would, however, work incorrectly, or an error would be reported by the compiler later, when attempting to explicitly recompile some dependent classes.

5. RELATED WORK

So far we have found only three products supporting smart, or incremental, compilation of Java applications (for comparison, the number of Java IDEs available now is nearly 20). Unfortunately, there are no publications on any of them, that would explain the internal details. Therefore the following discussion is based entirely on our own informal evaluation.

Borland JBuilder [7] Java IDE supports smart compilation (called "smart make" in its terminology), as one of the compiler options, along with two others: "recompile the current file" and "recompile the whole project". The implementation of "smart make" seems to be based on the same principles as our implementation: the so-called *dependency cache*, which is equivalent to our project database, is created for each project and then used to determine which classes need recompilation.

IBM VisualAge for Java [14] is another Java IDE which provides smart (*incremental*, in their terminology) compilation. In VisualAge, the developer edits, saves and recompiles methods individually, similar to the approach pioneered in Smalltalk, and is in contrast with most of the other IDEs which operate with the more traditional file-based source code representation. The incremental compiler is invoked automatically whenever a new Java class, interface, or method is created, or an existing one is changed. It recompiles the changed item, as well as all of the other items within the work space, that are affected by this change. If an incompatible change is made, the compiler immediately flags any inconsistencies created by that change.

Eclipse [2] is a highly extensible and configurable IDE (it is dubbed "a universal tool platform" by its creators), that is being developed as an open source project led by IBM. It has an incremental compiler, which is invoked every time a source file is changed and saved. The compiler checks any changed dependencies and flags all errors in sources that become inconsistent as a result of the change — much like in VisualAge.

The last product known to us that supports smart (incremental) compilation is the Java compiler called Jikes [13], an open source project being developed at IBM. This product is not an IDE, but a command-line compiler, which has an additional "incremental build" mode. The compiler enters this mode if it is invoked with a special "`++`" command-line option for a `.java` source file containing the `main(String args[])` method, e.g.

```
>jikes ++ myprog.java
```

This will compile `myprog.java` and other files that it depends on, as needed, and leave Jikes running. The developer can then change `myprog.java` or any of the source files it depends on, and simply hit "Enter" at the command line to tell Jikes to re-check the dependencies and only recompile files as required to bring the entire project up to date. The compiler will stay in this "hit Enter/rebuild" loop until the developer enters a "q", which tells it to terminate. This approach works well if the developer needs to just recompile the source files whenever they change, but, unfortunately, is unusable if this recompilation is a part of a more general authomated make process.

As for other programming languages and academic works, we have managed to find only one relatively recent work [9] — a PhD thesis by Crelier describing design and implementation of fine-grain dependency checking for Oberon. Oberon supports dynamic loading and linking of software modules, but these modules themselves are machine-dependent object files. Therefore, a part of the problem solved in that work was to provide sufficient type information for symbols (called objects in this work) exported by modules, in such a way that if a type of an exported object changes, the change is reflected in the exported type information. This was achieved by using *fingerprints* which were calculated as a function of, essentially, type name and structure information. Most of the complexity of this work lies in the area of correct fingerprint computation for complex type and their optimal storage in symbol files associated with modules. Change to the fingerprint for an exported object necessarily leads to recompilation of clients that import this object. This is probably correct for Oberon, given that it does not provide e.g. various levels of member accessibility, such as `protected` or package-private access, or abstract, interface and nested types. The latter are the cause of most of the non-trivial entries in our tables of source incompatible changes. Determining how to calculate a fingerprint for an object in Crelier's work seems quite similar to detecting an incompatible change and finding what can be affected by it, as it is done in our work.

6. FUTURE WORK

The tables of source incompatible changes and affected classes presented in Section 3 were defined using the JLS, other works, and our own knowledge and experience, but, unfortunately, there is no formal proof of their completeness and correctness. Consequently, we cannot be absolutely sure that these tables identify all of the affected classes for each incompatible change. There is more confidence that they cover all possible incompatible changes to Java classes, since, after all, the number of class elements is limited and the taxonomy of changes is created as a series of additions, removals and changes to these elements. Formal proof of completeness and correctness can probably be achieved using some kind of formal technique, e.g. similar to one used in [11] and other works by this group (see [1]).

It would also be very desirable to overcome the present problem with handling compile-time constants (see Table 3). At present, once such a constant changes, we have to recompile all classes that potentially can access it, because we have no way of finding out whether or not any class really uses this constant. The only way to fix this problem is to make the Java compiler more cooperative. However, just changing it so that it starts to put referenced compile-time constants into the constant pool, will probably not work very well. That is because our tool will then have to guess whether the compiler that it is currently using is a "new" or the "old" one. With many Java compilers on the market, each evolving in its own way, this may be not an easy task. So a better solution may be to introduce a special binary class attribute that should always be present in a class file created by a "new" compiler, and should contain information on compile-time constants referenced by a class, if there are any (or `null` otherwise).

One of our near-term goals is to provide an "incremental" execution mode in our tool, which should be of particular use when it is invoked from an IDE. In such a mode, Javamake reads the project database from disk once and then keeps it in memory permanently, until explicitly requested to terminate. Relatively high-cost database reading and writing operations are therefore performed only once each, upon the IDE startup and shutdown. Further speedup can be achieved if the IDE can provide Javamake the information on which sources have changed, so that the tool does not have to repeatedly perform another high-cost operation of checking all source file timestamps. Given that dependency checking itself is very fast (see Section 4), the "make" command of an IDE would impose virtually no overhead compared to the standard "compile" command.

7. CONCLUSIONS

Optimum build management is very important, even vital for developers of applications that consist of hundreds of thousands or millions lines of code. This issue is generally understood, and smart dependency checking (also called smart make or incremental compilation — all viewed by us as variants of the general smart recompilation technique) is provided in several IDEs and compilers for the Java programming language. Despite that, there are, to our best knowledge, no publications that document such a technology for the Java language (or any other statically type-checked programming language that compiles source code into portable bytecodes) at the level of detail that would allow to verify or improve it. This may be one reason why this technology does not seem to be implemented equally well in the few different tools where it is available. There are also very few contemporary academic publications in this area. With this paper, we hope to partially close this gap. We also hope that it may provide useful guidelines for designers of similar technologies for other programming languages.

8. ACKNOWLEDGEMENTS

The approach to smart dependency checking presented in this paper, and the first version of the Javamake technology, was first designed and implemented by the author as a part of his work on persistent class evolution in the PJama project [5, 10], that was developed jointly by the University of Glasgow, Scotland, and Sun Microsystems Laboratories. Professor Malcolm Atkinson, who was the author's supervisor, provided support and valuable input for this work. Later at Sun Microsystems, Mario Wolczko and Gilad Bracha both supported the general Javamake technology implementation effort, have read the early drafts of this paper and provided valuable comments. The author is also grateful to Mick Jordan, Greg Czajkowski, and numerous early adopters of Javamake, in particular Nascif Abousalh-Neto from Nortel Networks, who reported several bugs in its early releases and provided other useful feedback.

9. TRADEMARKS

10. REFERENCES

[1] Sound Languages Underpin Reliable Programming Project (SLURP), Imperial College, University of London, London, UK. http://www-dse.doc.ic.ac.uk/Projects/slurp/index.html.

[2] Eclipse Open Source Project. http://www.eclipse.org.

[3] NetBeans Open Source Project. http://www.netbeans.org.

[4] R. Adams, W. Tichy, and A. Weinert. The Cost of Selective Recompilation and Environment Processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, January 1994.

[5] M.P. Atkinson and M.J. Jordan. A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform. Technical report, Sun Microsystems Laboratories Inc and Department of Computing Science, University of Glasgow, UK, 2000. TR-2000-90, www.sun.com/research/forest/COM.Sun.Labs.Forest.doc.pjama_review.abs.html.

[6] Borland Inprise Inc. C++ Builder 5. http://www.inprise.com/bcppbuilder/.

[7] Borland Inprise Inc. JBuilder. http://www.inprise.com/jbuilder/.

[8] Borland Inprise Inc. Delphi 5. http://www.inprise.com/delphi/.

[9] R. Crelier. *Separate Compilation and Module Extension*. PhD thesis, ETH Zurich, Institute of Computer Systems, 1994.

[10] M. Dmitriev. *Safe Evolution of Large and Long-Lived Java Applications*. PhD thesis, Department of Computing Science, University of Glasgow, UK, 2001. Available at http://www.dcs.gla.ac.uk/∼misha/papers.

[11] S. Drossopoulou, D. Wragg, and S. Eisenbach. What *is* Java Binary Compatibility? In C. Chambers, editor, *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, Vancouver, BC, October 1998.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, June 2000.

[13] IBM Inc. The Jikes Open Source Project. http://oss.software.ibm.com/developerworks/opensource/jikes/project/index.html.

[14] IBM Inc. VisualAge for Java. http://www-4.ibm.com/software/ad/vajava/.

[15] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[16] Microsoft Corp. Microsoft Visual C++ Home Page. http://msdn.microsoft.com/visualc/default.asp.

[17] R.W. Schwanke and G.E. Kaiser. Smarter Recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4):627 – 632, October 1988.

[18] W. Tichy. Smart Recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986.

[19] WebGain. Visual Cafe 4.0. http://www.visualcafe.com/.