JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 11
Special issue: OOPS track at SAC 2004, Nicosia/Cyprus

Capturing ghost dependencies in Java sources

Giovanni Lagorio, Department of Computer Science of the University of Genova (DISI), Italy

All non trivial applications consist of many sources, which usually depend on each other. For this reason, a change to some sources may affect the compilation of other (unchanged) sources. Hence, the recompilation must be propagated to the unchanged sources that *depend* on the changed ones, in order to obtain the same result a global recompilation would produce.
Most IDEs (Integrated Development Environments) provide *smart* dependency checking; that is, they automate the task of finding these dependencies and propagating the recompilation when an application is rebuilt.
In this paper we study the problem of extracting dependency information from Java sources and propose an encoding of these dependency information as regular expressions. This encoding is both compact to store and fast to check.
Furthermore, our technique detects a particular kind of dependencies, which we call *ghost dependencies*, that popular tools, even commercial ones, fail to detect. Because of this failure, some required recompilations are not triggered by these tools and the result of their incremental recompilations is not equivalent to the recompilation of all sources.

1 INTRODUCTION

Most modern languages provide a mechanism called *separate compilation*, which avoids the need to recompile all sources each time a change is made.

Because sources usually depend on each other, recompiling only new/changed sources is not enough to obtain the same result a global recompilation would produce. A simple way to obtain the same result is to recompile, along new/changed sources, all unchanged sources which depend, directly or indirectly, on the new/changed ones.

This *cascading recompilation* mechanism, which is probably the most common form of *incremental recompilation*, is implemented by defining a dependency relation between source files. The most known tool for dealing with this kind of dependencies is *make* [3], which constructs a graph, based on an input file conventionally called *makefile*, and then performs a depth-first search of this graph to determine what work is necessary.

This solution usually triggers some useless recompilations, that is, recompilations which are not *necessary* in order to obtain the same result a global recompilation would produce. In previous work we have presented a compilation strategy for Java [1, 8, 9, 10] which is optimal from the theoretical point of view, but rather expensive to implement. An effective trade-off seems to be the use of such a strategy as a refinement of a less precise (but faster) strategy. For these reasons, in this paper we study how a cascading recompilation scheme can be applied to Java.

While in some languages, like C and C++, separate compilation requires the dependencies to be explicitly provided by the user[1], the Java mechanism allows the compiler to *infer* most of the dependencies. Indeed, inside a Java source file we do not find references to other source files, but only references to *names*.

Although at first sight Java rules for inferring dependencies might seem quite simple, they are not. Because the dot notation is used to name many different kinds of things (types, packages, fields and so on), its semantics is context dependent and tricky. This fact is sometimes surprising to programmers, who occasionally get unexpected name resolutions. Indeed, a peculiarity of Java is the fact that adding a *new* source may affect the compilation of unchanged sources, as a newly introduced type may alter the way names have to be interpreted.

We deal with this aspect by introducing a level of indirection in how dependencies are handled: instead of keeping information "source $\mathtt{S}_1$ depends on source $\mathtt{S}_2$" we keep information equivalent of "source $\mathtt{S}_1$ depends on set of names N". Hence, $\mathtt{S}_1$ depends on *any* source declaring names in N (since those declarations may affect the compilation of $\mathtt{S}_1$).

Failing to take this peculiarity into account prevents from providing a cascading recompilation equivalent to global recompilation. Yet, popular tools, like *Eclipse*, *Javamake* [2], *JBuilderX* and *Jikes*, seem to fail, according to the results of our tests. Indeed, as we show in the following by means of some examples, their incremental recompilations are *not equivalent* to global recompilation.

The rest of the paper is structured as follows. In Section 2 we recall how Java compilation works. In Section 3 we introduce a model of Java dependencies, considering both the apparent dependencies and the not-so-apparent ones, which we call *ghost dependencies*. By means of some examples, we show why incremental recompilations provided by most tools are not equivalent to global recompilation. In Section 4 we show how an incremental recompilation scheme, which takes all kinds of dependencies into account, can be built. In Section 5 we discuss implementation issues, in particular how regular expressions are perfectly suited for modeling ghost dependencies. Finally, in Section 6 we draw some conclusions.

2 HOW JAVA COMPILATION WORKS

In this section we recall how Java compilation works, emphasizing the issues which make inference of dependencies hard.

Consider the following example (for clarity, we always put at the beginning the

[1]By means of `#include` directives, which contain explicit filenames (although compiler switches like "-I" and "-L" complicate the matter further).

names of source files as single-line comments):

```
// File: A.java
class A {
    B aB ;
}
```

File `A.java` cannot be compiled in isolation, as it refers, in declaring its field `aB`, to a type named `B` whose declaration is unknown. If the binary file `B.class` is present, then it is used by the compiler to get the information about type `B`. Indeed, in Java a binary file is required to define exactly one type, say `T`, and to be named after it, that is, `T.class` (except that dots "." in type names are replaced by the character "$" in filenames).

Standard Java compilers take also the guess of relating the source file `B.java`, if present, with the binary file `B.class`. That is, when they need type information for a type called `B`, they look for both `B.java` and `B.class` and use the newer one (if the source is chosen, then it is recompiled too). Though this scheme seems reasonable, it does not work properly, as type `B`, if it is not `public`, can be declared in a file with *any* name; that is, `B.java` is as good as `everything_but_B.java`. The rationale for allowing declarations of non-public classes inside files with unrelated names is probably historical: before the introduction of inner/nested classes in JDK 1.1, allowing non-public classes to be in arbitrarily named files was a convenient loophole to allow to declare auxiliary and main classes together. For example, defining a `ListNode` class inside a file defining a (public) `List` class. Although programmers can (and should) now use inner/nested classes for such purposes, the file related rules were never changed, presumably for the sake of backwards compatibility.

In other words, there is a mismatch between sources and binaries, as each source can declare many types, while a binary can contain only one type. Furthermore, if a source file contains the declaration of a public type `T`, then the source has to be named `T.java`, otherwise there are no restrictions on the filename. On the other hand, as said before, a binary containing a type `T` has always to be named `T.class` (otherwise either it would not be found by the Java Virtual Machine or an exception would be thrown by the verifier).

Thus, the compilation of a source generates one or more binaries: each type declared in the source is translated into its own binary file. For instance, consider the following file:

```
// File: A_and_B.java
class A {}
class B {
   class Inner {}
}
```

The compilation of file `A_and_B.java` produces three files: `A.class`, `B.class` and `B$Inner.class`, corresponding to class `B.Inner`. Therefore, given the name of a type, say `C`, we can easily determine the name of its corresponding binary, `C.class`, but there is no way to determine which sources[2] might declare it (if the standard compiler cannot find `C.java`, then it assumes there is no source declaring `C` and gives up).

Note that if one of the top-level classes (that is, `A` or `B`) were `public`, then the file would have to be named after the public one, that is, `A.java` or `B.java`. This means that either `A` or `B` can be made `public`, renaming the source file too, but not both, unless their declarations are split in two distinct sources named `A.java` and `B.java`.

Many programmers are in the habit of declaring a single top-level type per file, despite the declared access modifier, naming the file after the type it declares. This is certainly a good convention, which we strongly recommend, but it is not a rule enforced by Java, so we cannot rely on this for finding which are the dependencies between Java sources. Relying on this convention, as Javamake [2] does, simplifies some issues but limits the applicability of the method.

For instance, consider the following files:

```
// File: A.java
class A {
   static double i = new B().j ;
   public static void main(String [] args) {}
}

// File: X.java
class B {
   int j = 0 ;
}

// File: Y.java
class B {
   double j = 0 ;
}
```

Both files `X.java` and `Y.java` declare a class named `B`. Each of them is self-contained, and they can be (separately) compiled into `B.class`. On the other hand, the file `A.java` declares a class named `A` and *refers to* class `B`. Because in these sources class `B` is declared twice, the compilation of all three sources *together* fails. Yet, using common compilers, as Sun's javac, we can *successfully* compile:

[2]Given a generic set of sources, more than one file might declare the same type; of course, that is an error condition and such sources cannot be compiled together.

- `X.java`, then `A.java`, then `Y.java`. In this case if we try to run `A`, then we get an exception `NoSuchFieldError` as `A` tries to access the `int` field `j` which is not present in class `B`. The point is that the binary file `B.class`, which `A.java` has been compiled with, has been replaced by the compilation of the *apparently unrelated* source `Y.java`.

- `Y.java`, then `A.java`, then `X.java`. This case is the opposite of the previous one: `A` cannot be run because the last compilation (of `X.java`) overwrote `B.class` (generated by the compilation of `Y.java`).

- `A.java` and `X.java` together, then `Y.java`. Same outcome of the first case.

- ...

Because the filename of a source hardly gives any information about which types are declared within, we need to parse each source file in order to know which types it declares.

3 THE MODEL

In this section we introduce a model of Java dependencies. This model allows us to describe precisely which are *all* the dependencies that have to be taken into account to obtain a cascading recompilation which is equivalent to a global recompilation.

Set

- $\mathcal{I}$ the set of all legal Java identifiers; that is, any non-empty sequence of letters and digits, which begins with a letter and is not a reserved keyword; note that the underscore "_" is considered a letter in Java. Examples of $id \in \mathcal{I}$ are: `A4`, `String` and `I_am_an_identifier`.

- $\mathcal{N}$ the set of all Java names; that is, non-empty sequences of dot separated identifiers. A *simple name* is a name without dots, that is, an identifier. Examples of $n \in \mathcal{N}$ are: `String`, `java.lang.String` and `myPackage.myClass`. In the following we use the metavariables t, p and n for, type, package and generic names respectively.

- $\mathcal{S}$ the set of all Java sources; that is, all strings which respect the grammar of Java.

We call

- *simple name* of a name its last identifier; that is, $\texttt{simpleName}(id_1.\cdots.id_{\text{n}}) = id_n$.

- *head* of a name its first identifier and *tail* the rest of the sequence; that is, $\texttt{head}(id_1.\cdots.id_n) = id_1$ and $\texttt{tail}(id_1.\cdots.id_n) = id_2.\cdots.id_n$. The tail of a single identifier is undefined, that is, $\texttt{tail}(id) = \bot$.

- $\texttt{provides}(s)$, for $s \in \mathcal{S}$, the set of top-level type names that s declares. We also write, abusing the notation, $\texttt{provides}(\texttt{aFilename.java})$ with the meaning $\texttt{provides}(\texttt{aFilename.java}) = \texttt{provides}(s)$ where s is the content of `aFilename.java` in the current compilation context (we formalize compilation contexts in Section 4).

As an example, given the following class declarations

```
// File: Foo.java
package p ;
class A {
   class Inner {}
}
class B {}
```

$\texttt{provides}(\text{Foo.java}) = \{\texttt{p.A, p.B}\}$. Inner types are not included in `provides` because their declaration is always syntactically enclosed inside the declaration of their outer type, hence an inner type t is recompiled if and only if the top-level type containing the declaration of t is recompiled.

We say that a source s_1 *directly depends* on another source s_2 if s_1 "refers to" a name contained in $\texttt{provides}(s_2)$; that is,

$$\texttt{dirDepends}(s_1, s_2) = \texttt{refersTo}(s_1) \cap \texttt{provides}(s_2) \neq \emptyset$$

We say that a source s_1 *depends* on s_n if there exists a chain $s_1, s_2, \ldots, s_n$ such that $\forall i \in \{1, \ldots, n-1\}$, s_i directly depends on s_{i+1}.

We have introduced the notion of dependency here because we believe it is quite intuitive and it should give the reader an idea of what we are aiming at, but this definition is incomplete until we give a precise meaning to the function `refersTo`. Packages, inner classes and `import` declarations complicate the model; we briefly illustrate why before introducing the formal definition of `refersTo`.

Packages and Inner classes

Packages are a means for grouping names in separate name spaces, which helps to prevent conflicts. If a type named t is declared inside a package `P`, then its fully qualified name is $\texttt{P}.t$ and the source file declaring t must reside in a directory called `P` (if a package is called `X.Y` then the file must reside in `Y`, which must be a subdirectory of `X`, and so on).

Inner classes are classes declared inside other classes; if a class `C` is declared inside a top-level class `Outer`, then the fully qualified name of `C` is `Outer.C`. If `Outer` is declared inside a package `P`, then the fully qualified names of `Outer` and `C` are, respectively, `P.Outer` and `P.Outer.C`.

This immediately shows a source of ambiguity: if a source s contains a name $n =$ `A.B.C`, what it refers to? If we knew n refers to a type, then the possibilities would be:

- `A` could be a top-level type (declared in the current package or in an explicitly imported one), `B` an inner type of `A` and `C` an inner type of `A.B`; in this case s would depend on (the sources declaring) `A`.
- `A` could be a package name, `B` a top-level type contained in `A` and `C` an inner type of `A.B`; in this case s would depend on (the sources declaring) `B`.
- `A.B` could be a package name, and `C` a top-level type contained in `A.B`; in this case s would depend on (the sources declaring) `C`.

Because we are interested in finding all sources a source s *might* depend on, we conservatively say that s depends on them all.

Moreover, if we do not know that the name `A.B.C` actually refers to a type, then there are even more possibilities, as just a *prefix* of `A.B.C` could be a type:

- `A.B.C` could be a package name. However, this case is not significant, as package names can appear in legal Java sources only at specific points (the package and import declarations) which we handle in a special way discussed below.
- `A` could be a package name, `B` a top-level type contained in `A` and `C` a non-private static member of `B`.
- `A` could be a top-level type (declared in the current package or in an explicitly imported one), `B` a non-private static member of `A` and `C` a non-private static member of the *type* of `B` (since Java allows accessing static fields of a type via any expression of that type — see 15.11.1 of [5]).

The last point is extremely peculiar and important: in that case `A.B` would be an *expression* of a certain type t, the declared type of field `B`, hence the source s would depend on *both*:

- the source declaring `A`;
- the source declaring t.

However, the latter dependency can be derived from the former, because if the source defining `A`, say $s_{\mathtt{A}}$, declares a field of type t, then $s_{\mathtt{A}}$ directly depends on the source defining t, say s_t. Of course, `B` could be an inherited field, but, in this case, $s_{\mathtt{A}}$ would directly depend on the file defining its direct superclass, say $s_{\mathtt{SuperA}}$, so the reasoning can be repeated there (up to the hierarchy, until the superclass declaring field `B` is found).

Analogously, if a source s contains a name `A.B.C.D.E`, then it could depend on the source defining `A`, `A.B`, `A.B.C`, and so on.

This reasoning brings us to the following definition, which models that the dependencies between sources are induced not only by names, but by their prefixes; that is, all the non-empty sequences that can be obtained from a name leaving out some identifiers at the end.

Given a name $n = id_1. \cdots .id_n$, set

$$\mathtt{prefixes}(n) = \{id_1,\ id_1.id_2,\ id_1.id_2.id_3,\ \ldots,\ id_1. \cdots .id_n\}$$

We also extend `prefixes` to set of names in the natural way:

$$\mathtt{prefixes}(\{n_1, \ldots, n_{\mathtt{k}}\}) = \bigcup_{i \in \{1,\ldots,k\}} \mathtt{prefixes}(n_{\mathtt{i}})$$

Do we really need to keep track of *all* these dependencies? After all, when a source s is compiled, all names it contains are resolved, hence we know what each name is *actually* referring to. Unfortunately, encoding just current dependencies would not work, as leaving out potential dependencies would mean not being able to respond correctly to all possible changes in other sources.

For instance, suppose to have the following files:

```
// File: A/B.java
package A ;

public class B {
   public int answer = 0 ;
}

// File: Test.java
class Test {
   public static void main(String [] args) {
      System.out.println(new A.B().answer) ;
   }
}
```

In this context the name `A.B` in `Test.java` refers to class `B` in package `A`, which is declared in the file `A/B.java`. Hence, excluding standard classes, that would be the only dependency of `Test.java`.

Yet, if we add the following source file:

```
// File: A.java
class A {
   static class B {
      public int answer = 42 ;
   }
}
```

and we recompile all three files, then the name `A.B` changes its meaning[3], referring to the nested class `B` of class `A`. So the only dependency of `Test.java` becomes `A.java`.

Hence if, in the first place, we had set the only dependency of `Test.java` to be `A/B.java`, then afterward we could not detect that `Test.java` has to be recompiled due to the addition of `A.java`.

This point is tricky: of course, before introducing `A.java` it does not make any sense to say that `Test.java` depends on `A.java`; even more, any file declaring `A` may affect the compilation of `Test.java`. In summary, besides actual dependencies, like the one between `Test.java` and `A/B.java`, there are "hidden" potential dependencies, which we call *ghost dependencies*, between `Test.java` and all files, *existing or future*, which declare something that may affect the compilation of `Test.java`.

We deal with this aspect by introducing a level of indirection in how dependencies are handled: instead of keeping information "source $\mathtt{S}_1$ depends on source $\mathtt{S}_2$" we keep the information "source $\mathtt{S}_1$ depends on a set of names N". Hence, $\mathtt{S}_1$ depends on *any* source s declaring names in N (since those declarations may affect the compilation of $\mathtt{S}_1$).

Failing to take ghost dependencies into account prevents from providing a cascading recompilation equivalent to global recompilation. Yet, popular tools seem to make this exact mistake and do not get this kind of dependency right:

- *Eclipse 2.1.3* (http://www.eclipse.org/),
- *Javamake 1.3.2* (http://www.experimentalstuff.com/Technologies/JavaMake/),
- *JBuilder X* (http://www.borland.com/jbuilder/) and
- *Jikes*[4] *1.21* (http://www-124.ibm.com/developerworks/opensource/jikes/),

[3]When a name can be interpreted as both a type name and a package name, the former interpretation is chosen — see 6.5.4.1 of [5].

[4]We have tested the "++" incremental compilation. Re-generating a makefile, using `-M`, after the addition of `A.java` works correctly, of course.

the latest available at the time of writing, do not recompile `Test.java` after the addition of `A.java`, so the result of their recompilations is *not equivalent* to the recompilation of all sources.

Import declarations

Let us now consider `import` declarations. There are two kinds of import declarations: *single type* and *on demand*. In the former case, a fully qualified type name is specified. The effect of such declarations is to make available the types they specify as simple names. For instance, after the following declaration:

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

the simple name `Vector` refers to `java.util.Vector` (when it is not *shadowed* or *obscured* by other declarations contained in the same file, see 6.3.1 and 6.3.2 of [5]).

So, a single type import of a name n_1 may affect the resolution of a name n_2 if $\texttt{simpleName}(n_1) = \texttt{head}(n_2)$ as, in this case, n_2 can stand for $n_1.\texttt{tail}(n_2)$, when $\texttt{tail}(n_2) \neq \perp$, or for n_1, when $\texttt{tail}(n_2) = \perp$. Note that it is an error to have two distinct single type import declarations for the same simple name[5], so single type import declarations cannot create ambiguity by themselves.

The second kind of import declaration is also known as *star import*. This kind of declaration imports all the public names of a package or type, making them available, for the compilation unit, as simple names. For instance,

```
import java.util.*;
```

allows to use simple names as `Vector` or `List` instead of their corresponding fully qualified names `java.util.Vector` and `java.util.List`.

So, after an import declaration `import` n`.*;` the effect is that all names n' in type declarations[6] can be interpreted both as n' and $n.n'$.

Star imports are another source of ghost dependencies. If two packages, say p_1 and p_2, are imported, then any name n can be interpreted, in addition to simply n, as $p_1.n$ and $p_2.n$. This fact is exploited in the following example, which is based on the example given by Todd Turnidge in Susan Eisenbach's web page http://www.doc.ic.ac.uk/~sue/packages.html.

[5]Unless they refer to the same type, in which case one of them is simply ignored — see 7.5.1 of [5].

[6]Import declarations have no effect on each other.

```
// File P1/A.java
package P1 ;
public class A {}

// File P2/Foo.java
package P2 ;
class Foo {}

// File Test.java
import P1.* ;
import P2.* ;

class Test {
   A anA ;
}
```

These source files can be successfully compiled and it is easy to see that `Test.java` depends on `P1/A.java`, as the former names `A` which is resolved to `P1.A`, declared by the latter. Note that class `Foo` is not really required to make our point, but we had to put its source into directory `P2` to make `javac` happy (otherwise we get the error "package P2 does not exist") — other compilers do not require this.

If we add now the following file:

```
// File P2/A.java
package P2 ;
public class A {}
```

then the compilation of `Test.java` becomes undefined, as the simple name `A` is now ambiguous (it could be both `P1.A` and `P2.A`).

We have tried this example with the aforementioned tools and, in this case, only *Eclipse* correctly finds the ambiguity. The other tools do not propagate the compilation of the new file `P2/A.java` to the unchanged file `Test.java`, so their incremental recompilations terminate successfully.

Formal Definition

Now that we have seen all the meanings a name can have, we can formalize what "a source *refers to* a name" means.

Definition 1 *If a source file $s \in \mathcal{S}$*

- *declares types inside package $p \in \mathcal{N} \cup \{\epsilon\}$, where ϵ represents the name of the unnamed package (we define $\epsilon.n = n$) ;*
- *imports the names $N_{OnDemand} \subseteq \mathcal{N}$ by on demand import declarations;*
- *imports the names $N_{SingleT} \subseteq \mathcal{N}$ by single type declarations;*
- *contains the (neither obscured nor shadowed) names $N_{Body} \subseteq \mathcal{N}$ in the body.*

Then we say that s refers to *all names in the set*

$$\texttt{refersTo}(s) = R_{Decls} \cup R_{OnDemand} \cup R_{SingleT}$$

where:

- $R_{Decls} = \texttt{prefixes}(N_{OnDemand}) \cup \texttt{prefixes}(N_{SingleT})$
- $R_{OnDemand} = \{n_1.n_2 | n_1 \in N_{OnDemand} \cup \{p\},\ n_2 \in \texttt{prefixes}(N_{Body})\}$
- $R_{SingleT} = \{n_1.\texttt{tail}(n_2) | n_1 \in N_{SingleT},\ n_2 \in \texttt{prefixes}(N_{Body}),\ \texttt{simpleName}(n_1) = \texttt{head}(n_2),\ \texttt{tail}(n_2) \neq \perp\}$

That is,

- R_{Decls} contains just the prefixes of imported names because import declarations have no effect on each other.
- $R_{OnDemand}$ contains all combinations of a star imported name and the prefixes of names contained in the body. Note that the current package is considered implicitly star imported.
- $R_{SingleT}$ contains all combinations of single type imported names and the tails of names in the body that are affected by the corresponding import declaration. Names which are single identifiers (that is, the ones such that $\texttt{tail}(n_2) = \perp$) are not included here because their "expansion" due to a single type imported name n corresponds to name the n itself, hence it is contained in $N_{SingleT}$ ($\subseteq R_{Decls}$).

4 AN INCREMENTAL RECOMPILATION SCHEME

In this section we show how function `dirDepends` can be used to build an incremental recompilation scheme; that is, how we can determine which files have to be recompiled after some changes.

First of all, let us recall the definition of `dirDepends` (from page 82)

$$\texttt{dirDepends}(s_1, s_2) = \texttt{refersTo}(s_1) \cap \texttt{provides}(s_2) \neq \emptyset$$

and introduce *compilation contexts*, which model the fact that sources are stored in a filesystem. A compilation context is a (partial) map from filenames to their corresponding sources; if $\mathcal{F}$ is the set of all filenames, ranged over by f, then a compilation context is:

$$cc : \mathcal{F} \to \mathcal{S}$$

Given a compilation context cc we derive from `dirDepends` the following new relation:

$$\to_{cc} = \{(f_1 \to_{cc} f_2) \,|\, \texttt{dirDepends}(s_1, s_2),\ s_1 = cc(f_1),\ s_2 = cc(f_2)\}^+$$

with the meaning that if $(f_1 \to_{cc} f_2)$ holds, then the compilation of f_1 may be affected by a change in f_2 (the $\_^+$ denotes the transitive closure).

We also define the set

$$\texttt{requires}(f, cc) = \{f' | (f \to_{cc} f')\}$$

of all sources a file f depends on in a compilation context cc. Conversely, all sources that depends on f in cc are:

$$\texttt{requiring}(f, cc) = \{f' | (f' \to_{cc} f)\}$$

Given two compilation contexts $cc_{\texttt{old}}$ and $cc_{\texttt{new}}$, the set of (new and) changed files is defined as:

$$\texttt{changed}(cc_{\texttt{old}}, cc_{\texttt{new}}) = \{f | cc_{\texttt{new}}(f) \neq \bot, cc_{\texttt{old}}(f) \neq cc_{\texttt{new}}(f)\}$$

The condition $cc_{\texttt{old}}(f) \neq cc_{\texttt{new}}(f)$ covers also the case where a new filename has been introduced in $cc_{\texttt{new}}$, as in this case $cc_{\texttt{old}}(f) = \bot$. We give an abstract definition here; of course, an implementation would derive which files are new or changed storing, and then comparing, timestamps.

The set of deleted files is defined as:

$$\texttt{deleted}(cc_{\texttt{old}}, cc_{\texttt{new}}) = \{f | cc_{\texttt{old}}(f) \neq \bot,\ cc_{\texttt{new}}(f) = \bot\}$$

Of course, if the current compilation context is $cc_{\texttt{new}}$ and the last compilation was successfully performed in compilation context $cc_{\texttt{old}}$, then all the sources in $\texttt{changed}(cc_{\texttt{old}}, cc_{\texttt{new}})$ need to be (re)compiled. Let us now analyze which *possibly unchanged* sources have to be recompiled as well.

For each new/changed or deleted file $f \in \texttt{changed}(cc_{\texttt{old}}, cc_{\texttt{new}}) \cup \texttt{deleted}(cc_{\texttt{old}}, cc_{\texttt{new}})$ we trigger the (re)compilation of all files that might depend on it:

$$(\texttt{requiring}(f, cc_{\texttt{old}}) \cup \texttt{requiring}(f, cc_{\texttt{new}})) \setminus \texttt{deleted}(cc_{\texttt{old}}, cc_{\texttt{new}})$$

that is:

- the sources that depended on f in the old context — these are the "standard" recompilations found by existing tools;
- the sources that *currently* depends on f — this is needed for dealing with *ghost dependencies*;
- . . . as long as these sources are still there ☺ .

Note that generally $\texttt{requiring}(f, cc_{\texttt{old}}) \not\subseteq \texttt{requiring}(f, cc_{\texttt{new}})$, because some declarations contained within f or f itself could have been removed. On the other hand, in addition to trivial cases where f has changed, the inclusion $\texttt{requiring}(f, cc_{\texttt{old}}) \supseteq \texttt{requiring}(f, cc_{\texttt{new}})$ does not hold even when $cc_{\texttt{old}}(f) = cc_{\texttt{new}}(f) = s$, as we showed in previous examples. Indeed, this (wrong) assumption is what makes the examined tools fail in our examples.

5 IMPLEMENTATION

In this section we consider how to implement:

- the function `provides`;
- the function `refersTo` and
- the test $\texttt{refersTo}(s_1) \cap \texttt{provides}(s_2) \neq \emptyset$.

In implementing functions `provides` and `refersTo`, two things are to be considered. The former is how to collect all the names, which is an easy task. The latter is how to represent these sets of names, using an encoding which is compact but allowing a fast intersection test (needed by `dirDepends` on which the incremental recompilation scheme is built).

The sets of names corresponding to both functions, $\texttt{provides}(s)$ and $\texttt{refersTo}(s)$, can be easily found visiting the syntax tree of s. *Javac*, for instance, uses the pattern *visitor* [4], which permits to add new kinds of visit without changing the classes for representing syntax trees. Because sets $\texttt{provides}(s)$ and $\texttt{refersTo}(s)$ can change only when s changes, the parsing steps needed to keep their representation up to date are basically free. Indeed, we need to parse only new and changed sources, that is, those sources that would be parsed anyway (in any incremental recompilation mechanism) to be (re)compiled.

However, the most interesting part of implementation is the way sets are represented. Using regular expressions [7] we found an encoding which is both compact and allows to quickly[7] check whether two sets intersects.

[7]Although we do not have actual benchmarks yet, our assumption is based on the fact that matching regular expressions is *fast*.

Given a set of names N we denote with $[\![N]\!]_{\mathtt{S}}$ its representation as a string, defined as

$$[\![\{n_1, \ldots, n_k\}]\!]_{\mathtt{S}} = \#n_1\# \ldots \#n_k\#$$

That is, we concatenate[8] all names into a single string, keeping them separate by the special character "#" (any character which cannot be contained in a name would do).

Given a set of names N, we denote with $[\![N]\!]_{\mathtt{R}}$ its representation as regular expression (we discuss the details of this translation below).

Given two sources, say s_1 and s_2, we encode the set $\mathtt{provides}(s_2)$ with the string $\mathtt{str} = [\![\mathtt{provides}(s_2)]\!]_{\mathtt{S}}$ and the set $\mathtt{refersTo}(s_1)$ with the regular expression $R = [\![\mathtt{refersTo}(s_1)]\!]_{\mathtt{R}}$. With these representations, the test $\mathtt{refersTo}(s_1) \cap \mathtt{provides}(s_2) \neq \emptyset$ corresponds to checking whether a substring of the string `str` matches the regular expression R.

For readability, we enclose regular expressions between "$\ll$" and "$\gg$" and use the syntax of Java regular expressions [6] except that we pretend the period is not a metacharacter and do not escape it (that is, we use just $\ll.\gg$ instead of $\ll\backslash.\gg$ to represent the single character ".").

Because, for any s, the set $\mathtt{refersTo}(s) = \{n_1, \ldots, n_k\}$ is finite, we could represent it using a long alternation $\ll n_1|n_2|\ldots|n_k \gg$, but such an encoding would be rather space consuming. Since one of our goals is to keep the representation as compact as possible, we have examined the pieces that make up $\mathtt{refersTo}(s)$ trying to find the best encoding for each case.

For instance, sets produced by function `prefixes` can be compactly represented using the quantifier "?" (which stands for "once or not at all").

Given a name $n = id_1.id_2. \cdots .id_n$, then we define $[\![\mathtt{prefixes}(n)]\!]_{\mathtt{R}}$ as:

$$\ll id_1(.id_2(.id_3(\cdots(.id_n)?\cdots)?)?)? \gg$$

For instance, $[\![\mathtt{prefixes(A.B.C.D)}]\!]_{\mathtt{R}} = r$, with $r = \ll \mathtt{A(.B(.C(.D)?)?)?} \gg$. Let us decompose r from the outside inward: $\ll \mathtt{A}X\mathtt{?} \gg$ means "`A` or `A` followed by X"; in our case $X = \ll \mathtt{.B(.C(.D)?)?} \gg$ which we can write as $X = \ll \mathtt{.B}Y? \gg$. Hence, X means "`.B` or `.B` followed by Y", so the set r describes is:

- `A` or
- `A` followed by:
  - `.B`, that is, `A.B` or
  - `.B` followed by Y, that is, `A.B`Y

[8]In any order (the order is immaterial because of the way we use these strings).

Repeating this reasoning on Y and so forth, we can see that r encodes all and only the elements of `prefixes`(A.B.C.D).

Set unions are translated using the alternation operator "|", that is,

$$[\![S_1 \cup S_2]\!]_{\mathtt{R}} = \ll [\![S_1]\!]_{\mathtt{R}} | [\![S_2]\!]_{\mathtt{R}} \gg$$

Before defining the translation of set $\mathtt{refersTo}(s) = R_{Decls} \cup R_{OnDemand} \cup R_{SingleT}$, defined in Definition 1, let us consider the translation of its subsets R_{Decls}, $R_{OnDemand}$ and $R_{SingleT}$ first. In what follows we assume to be in the context of Definition 1, so we do not repeat the meaning of variable names here.

The translation of set R_{Decls} is straightforward:

$$[\![R_{Decls}]\!]_{\mathtt{R}} = \ll [\![\mathtt{prefixes}(N_{OnDemand})]\!]_{\mathtt{R}} \mid [\![\mathtt{prefixes}(N_{SingleT})]\!]_{\mathtt{R}} \gg$$

The translation of set $R_{OnDemand}$ depends on p; if $p = \epsilon$, that is, type declarations are contained in the unnamed package, the translation is:

$$[\![R_{OnDemand}]\!]_{\mathtt{R}} = \ll ([\![N_{OnDemand}]\!]_{\mathtt{R}} . |) [\![\mathtt{prefixes}(N_{Body})]\!]_{\mathtt{R}} \gg$$

otherwise it is:

$$[\![R_{OnDemand}]\!]_{\mathtt{R}} = \ll ([\![N_{OnDemand}]\!]_{\mathtt{R}} | p). [\![\mathtt{prefixes}(N_{Body})]\!]_{\mathtt{R}} \gg$$

The translation of set $R_{SingleT}$ is more complex. Given $N_{SingleT} = \{n_1, \ldots, n_k\}$, we partition $R_{SingleT}$ into:

$$R_{SingleT} = \bigcup_{i \in \{1,\ldots,k\}} S_i, \quad \text{where:}$$

$$S_i = \{n_i.\mathtt{tail}(n) | n \in \mathtt{prefixes}(N_{Body}),\ \mathtt{simpleName}(n_{\mathtt{i}}) = \mathtt{head}(n), \mathtt{tail}(n) \neq \bot\}$$

In order to obtain a more compact translation, we rewrite S_i as:

$$\begin{aligned} S_i = \ & \{n_i.\mathtt{tail}(n) | n \in \mathtt{prefixes}(\{n \in N_{Body},\ \mathtt{simpleName}(n_{\mathtt{i}}) = \mathtt{head}(n), \mathtt{tail}(n) \neq \bot\}) = \\ & \{n_i.n | n \in \mathtt{prefixes}(\mathtt{tail}(\{n \in N_{Body},\ \mathtt{simpleName}(n_{\mathtt{i}}) = \mathtt{head}(n), \mathtt{tail}(n) \neq \bot\})) \end{aligned}$$

The former simplification corresponds to calculating the prefixes after having selected the names, instead of calculating all prefixes and then selecting the names (because the selection is driven by the head of the names, the order of these two operations is immaterial). The latter simplification corresponds to taking the prefixes of the tails instead of vice versa; because of the condition $\mathtt{tail}(n) \neq \bot$, which avoids the undefined expression $\mathtt{prefixes}(\bot)$, the order is immaterial in this case too.

Hence,

$$[\![S_i]\!]_{\mathtt{R}} = \ll n_i. [\![\mathtt{prefixes}(\mathtt{tail}(\{n \in N_{Body},\ \mathtt{simpleName}(n_{\mathtt{i}}) = \mathtt{head}(n), \mathtt{tail}(n) \neq \bot\}))]\!]_{\mathtt{R}} \gg$$

and

$$[\![R_{SingleT}]\!]_{\mathtt{R}} = \ll [\![S_1]\!]_{\mathtt{R}} | [\![S_2]\!]_{\mathtt{R}} | \ldots | [\![S_k]\!]_{\mathtt{R}} \gg$$

The topmost translation requires a little adjustment to work correctly; that is, the whole set $\texttt{refersTo}(s) = R_{Decls} \cup R_{OnDemand} \cup R_{SingleT}$ is translated into:

$$\ll \# [\![R_{Decls}]\!]_{\mathtt{R}} | [\![R_{OnDemand}]\!]_{\mathtt{R}} | [\![R_{SingleT}]\!]_{\mathtt{R}} \# \gg$$

In this case we have to delimit the final expression with a couple of "#"s. These characters ensure that matches of partial names do not influence intersection tests. Consider, for instance, $\texttt{provides}(s_1) = \{\texttt{A.B.C}\}$ and $\texttt{refersTo}(s_2) = \{\texttt{B}, \texttt{B.C}\}$. Although their intersection is clearly empty, if we omitted the delimiting "#"s then we would get two unwanted matches (wrongly representing a non-empty intersection). Indeed,

- $[\![\{\texttt{A.B.C}\}]\!]_{\mathtt{S}} =$ "`#A.B.C#`" and
- $R = [\![\{\texttt{B}, \texttt{B.C}\}]\!]_{\mathtt{R}} = \ll \texttt{B}(\texttt{.C})? \gg$.

Hence, R has two matches: "`#A.`$\boxed{B}$`.C#`" and "`#A.`$\boxed{B.C}$`#`". With the delimiters, instead, we force R to match only whole names and we get no unwanted partial matches.

6 CONCLUSIONS

This paper can be seen as a self-contained analysis of dependencies in Java, but also as a follow-up of our previous work [1, 8, 9, 10], where we have given a type system, for a substantial subset of Java, that can be fruitfully used to implement a selective recompilation strategy. Our strategy is both *sound* and *minimal*, that is, its result is equivalent to global recompilation of all sources; yet, our strategy never triggers useless recompilations (i.e., recompilations which produce binaries equal to the existing ones).

While from a theoretical point of view this was the best we could achieve, from a practical point of view the handling of the full Java language requires evaluating rather complex type assumptions, which is time-consuming. So, a more effective way to apply those ideas is to use such a recompilation strategy as a refinement of some other less precise (but faster) strategy; for instance, as a refinement for a cascading recompilation strategy. This was the motivation for this paper.

On the one hand, a cascading recompilation scheme is both easier to implement and more efficient (in selecting which sources have to be recompiled) than the solution based on type assumptions, but, on the other hand, a cascading recompilation scheme triggers useless recompilations in most cases, since it cannot tell apart changes that affect the compilation of dependent sources from changes that do not.

While there is already some work on cascading recompilation schemes for Java [2], our tests show that popular tools (Eclipse, Javamake, JBuilderX and Jikes) fail to trigger some recompilations which are required to obtain the same result a global recompilation would produce.

The source of these problems seems to be the fact that the introduction of a *new* source may change the result of the (re)compilation of *unchanged* sources. In order to deal with these *ghost dependencies* we have introduced a level of indirection in handling dependencies and shown how regular expressions can be used to encode these dependencies.

Our encoding is compact and allows to quickly determine which new dependencies have been introduced by the addition or the modification of some files.

Acknowledgments We warmly thank Elena Zucca and the anonymous referees for their useful suggestions and feedback.

This work has been partially supported by Dynamic Assembly, Reconfiguration and Type-checking - EC project IST-2001-33477, and APPSEM II - Thematic network IST-2001-38957.

REFERENCES

[1] D. Ancona and G. Lagorio. Stronger Typings for Smarter Recompilation of Java-like Languages. *Journal of Object Technology*, 3(6):5–25, June 2004. Special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs.

[2] M. Dmitriev. Language-specific make technology for the Java programming language. *ACM SIGPLAN Notices*, 37(11):373–385, 2002.

[3] S. I. Feldman. Make - A Program for Maintaining Computer Programs. *Software - Practice and Experience*, 9(4):255–65, 1979.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

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

[6] Scott A. Hommel. *Regular Expressions – online trail of The Java Tutorial (available at http://java.sun.com/docs/books/tutorial/extra/regex/index.html)*. Sun Microsystems.

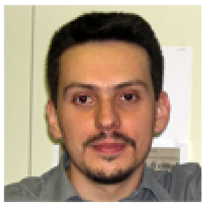
[7] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, New Jersey, 1956.

[8] G. Lagorio. Towards a smart compilation manager for Java. In Blundo and Laneve, editors, *Italian Conf. on Theoretical Computer Science 2003*, number 2841 in Lecture Notes in Computer Science, pages 302–315. Springer, October 2003.

[9] G. Lagorio. Another step towards a smart compilation manager for Java. In Hisham Haddad, Andrea Omicini, Roger L. Wainwright, and Lorie M. Liebrock, editors, *ACM Symp. on Applied Computing (SAC 2004), Special Track on Object-Oriented Programming Languages and Systems*, pages 1275–1280. ACM Press, March 2004.

[10] G. Lagorio. *Type systems for Java separate compilation and selective recompilation.* PhD thesis, University of Genova, May 2004.

ABOUT THE AUTHORS



Giovanni Lagorio took a Ph.D. in Computer Science at the University of Genova on May 2004. His research interests are in the area of programming languages; in particular, design and foundations of modular and object-oriented languages and systems. He can be reached at lagorio@disi.unige.it.
See also http://www.disi.unige.it/person/LagorioG/.