A Case For Sealing Classes In Java*

Marina Biberstein biberstein@il.ibm.com

Vugranam C. Sreedhar sreedhar@watson.ibm.com

Ayal Zaks zaks@us.ibm.com

May 9, 2002

Abstract

It is a well-known fact that inheritance as defined in most existing object-oriented languages breaks encapsulation in a very subtle way. For instance, Java provides facilities for encapsulation both at the class-level and at the package-level. But it also introduces language constructs (such as `public`/ `protected` modifier and inheritance) that lets clients to access the internals of a class in a subtle way.

In this paper we introduce the notion of *class sealing* in Java that selectively prevent clients of a package from accessing the internals of a class via inheritance. Class sealing allows extension of classes within a package, but prevent clients from extending such classes outside of the package. We will discuss three areas of application of class sealing: maintenance, security, and program optimization. We provide empirical evidence, collected from large bodies of real-life code, showing that sealed classes fit well within existing coding practice.

1 Introduction

In recent years the Java programming language has achieved widespread acceptance for developing large-scale applications. Java encourages the notion of application development using "off-the-shelf" units by providing a large collection of libraries. The popularity of Java has risen because library units can be developed on one platform, transported over the Internet, and be deployed on a different platform. Ideally, library units should be reusable, secure, and optimized for performance. A key concept for achieving reusability, security, and optimization of library units is *encapsulation*. It is well-known that one cannot achieve both true encapsulation and *unrestricted* class inheritance with overriding capabilities [7]. To be able to override a method in a class one should be able to understand the internal details of the class. Current object-oriented (OO) languages, such as Java, provide language constructs (such as `public/protected` modifiers and inheritance) for subclasses to access the internal details of their ancestor classes. These language constructs clearly breaks encapsulation.

The problem of encapsulation is particularly serious for Java due to the presence of open packages and dynamic class loading. In Java, a package is a collection of related classes and interfaces that provide access protection and namespace management. Packages allow Java programmers to develop independent software units that can be easily deployed, distributed as off-the-shelf components, and enforce an encapsulation scope that is larger than a class but smaller than a complete application. Unlike ML and Ada modules, Java packages are *open* entities: at any time during the program execution, a new class or interface can join the package simply by claiming to be its member. The drawbacks of open packaging include poor encapsulation, security holes, unstable software deployment, conservative program analysis and optimizations.

Another factor that aggravates the problem of having open packages is the support for dynamic extension of classes during execution via dynamic class loading. Java developers have exploited dynamic class loading to develop flexible and robust applications. For instance, Remote Method Invocation (RMI) uses dynamic class loading to seamlessly migrate objects between virtual machines without having to distribute the supporting class files. Dynamic class loading is closely tied to the concept of polymorphism and dynamic method binding. In polymorphism, a reference to an object can be bound to any compatible object-type during program execution, and dynamic class loading lets the set of compatible object-types grow dynamically during program execution. The notion of dynamic growth of compatible

* **Word count: 5907**

object types gives developers the needed flexibility, allowing the component developer, or indeed the component user, to choose implementations of the component's interfaces during configuration or run time.

Open packages and unrestricted dynamic extensions introduce serious problems for application development, software maintenance, optimizations, and security. Java 2 amended open packages by providing the *package sealing mechanism*.[1] Sealing guarantees that all package members originate from the same JAR file, thus restricting package membership at run-time. This seemingly modest change is in the right direction for creating a true encapsulation unit, and makes a sealed package the largest unit whose integrity at run-time can be guaranteed by the system. In Java, where different packages can be loaded at run-time from different sources, careful packaging and maximal in-package encapsulation becomes vital to avoid security, design, and deployment hazards. Sealed packages still do not solve the problem of encapsulation and inheritance. This is because an outside intruder can still subclass (i.e., extend) a `public` class in a sealed package. Although Java SDK 1.2 took a step in the right direction by defining package sealing, it did not go far enough to make sealing a first-class concept, and did not provide fine grain control over the members of sealed packages.

In this paper we introduce the concept of *sealed classes* to Java. Sealed classes allow a flexible and explicit way of controlling extension of classes. Sealing of classes essentially prevents (unrestricted) extension of classes outside of a package, but allows extensions of classes within the package. We will show how class sealing can be used to achieve a "statically bounded" form of polymorphism, called *oligomorphism* [4]. Oligomorphism has several benefits. First, it allows one to perform better program optimization by statically resolving virtual method dispatch. Second, it improves encapsulation, because we are guaranteed that the set of types is bound statically. Also, it improves security—a rogue program cannot add new types during execution and masquerade as instances of the "innocent" base class. Finally, oligomorphism also encourages software developers to turn packages into "almost components", leading to better programming style and helping to avoid maintenance hazards such as the "fragile base class problem" [7].

Our motivation for sealing classes is based on the following simple observation. In Java, the modifier `public` of a class in a package has an *overloaded semantics*. A `public` class in a given package can be both accessed and extended (i.e., sub-classed) by other classes from any package. Many times a programmer wants to allow clients (of a package) to access a certain class while restricting its subclassing. This can currently be achieved only by declaring the class to be `final`, thereby disallowing any extension of the class, even from within the package. In other words, there is no way in Java to forbid the extension of a `public` class outside its package, and yet allow extension of the same class within its package. Sealing classes introduce the flexibility of preventing the extension of a class hierarchy outside a package, but allowing class extensions within the package. In this paper, we also provide preliminary experimental results that gives insight on the effect of class sealing on typical Java libraries and applications.

The rest of the paper is organized as follows: Section 2 introduces the concept of class sealing and oligomorphism. Section 3 discusses how sealing can help in improving encapsulation. Section 4 discusses application of class sealing and oligomorphism in the context of program analysis and optimization. Section 5 presents some preliminary experiments on class sealing. Section 6 discusses related work and some finer issues of class sealing. Finally, Section 7 gives our conclusion.

2 Sealed Classes and Polymorphism

In this section we will first briefly discuss Java sealed packages (Section 2.1). Next we introduce the concepts of polymorphism and oligomorphism (Section 2.2). We will finally discuss how to seal classes and point out some finer details of class sealing (Section 2.3).

2.1 Sealed Packages in Java

Starting from the SDK 1.2, Java platforms introduced the concept of *sealing* JAR files and packages within JAR files. A *sealed package* ensures that all classes defined in that package originate from the same JAR file. A `SecurityException` is thrown if this restriction is violated at runtime. Sealing is specified via a new `Manifest` file attribute. For example,

```
Name: com/ibm/Vehicle/
Sealed: true
```

[1] See `http://java.sun.com/products/jdk/1.2/docs/guide/extensions/spec.html`.

specifies that the `com.ibm.Vehicle` package is sealed.[2]

Introduction of sealing was originally motivated by security considerations. Package sealing prevents a rogue program from adding its classes to a package, thereby acquiring access to package-protected members of the package. Another important mechanism provided by Java is the ability to *sign* JAR files. This can be used to guard against attempts to modify a JAR file by unauthorized parties, thus complementing the package sealing mechanism.

2.2 Polymorphism and Oligomorphism

Polymorphism is the ability to take many forms. In Java, polymorphism is achieved by sub-classing and inheritance. During execution, a method invocation such as $p.foo()$ is transparently dispatched to one of the subclasses of the declared type of p. In Java, additional subclasses could be dynamically loaded at runtime, and thereby dynamically extending the class hierarchy. But sometimes you want to bound statically the set of types that a reference can point to during execution. One can identify a "bounded" sort of polymorphism, called *oligomorphsim*, which takes only finite sort of forms [4]. A class is *oligomorphic* if the class hierarchy underneath it can be statically bound (i.e., set of descendent subclasses of a class is known to the language processor) [4]. Notice that if the declared type of a reference variable is oligomorphic, then the set of runtime type of the objects that the reference variable can potentially point to is also statically bound. In Java, one can achieve a limited form of oligomorphism by using the modifier `final` for classes, or by hiding the class hierarchy within the package (via a package-level protection mechanism) [4]. In the next section we will discuss how to achieve oligomorphism that is more flexible than `final` classes.

2.3 Sealed Classes

Throughout our discussion we will assume (unless otherwise explicitly stated) that packages are *sealed* within *signed* JAR files.[3] A `public` class in a package has an overloaded semantics. First, it means that a client can use the class as a type (such as in method parameters, return type, and construction of new instances of the class). Second, a client can also `extend` the public class. Sometimes, however, you only want to allow a client to use a class as a type, while restricting its extensions. In Java, one way to do this is to use the modifier `final`, but this will forbid extension of a class within its own package. To better cope with this situation, we introduce the concept of *sealed* classes.

Definition 2.1 *A class C (or an interface I) in a package P is said to be* **sealed** *if the class hierarchy underneath C (or I) is bounded to be within P.*

We can make the following simple observation for sealed classes.

Property 2.2 *If a class is sealed then it is oligomorphic.*

It is important to keep in mind that oligomorphism is an abstract concept while class sealing is a linguistic mechanism for achieving oligomorphism. We can easily see that if a sealed class is in an open package, then it is not necessarily oligomorphic. Note also that while non-public classes cannot be directly extended by classes outside the package, they can be extended by public classes within the package, which, in turn, can be extended by classes in other packages. Thus sealing is applicable to both public and package-private classes. A `SecurityException` can be thrown (as in the sealed package case) when a client attempts to extend a sealed class.

Oligomorphism has several benefits. First, if we know that a class is oligomorphic, we can perform better static analysis and optimize messages sent to this class. Second, it improves encapsulation, because we are guaranteed that the class hierarchy underneath an oligomorphic class in a (sealed) package will not be extended at runtime. Finally, it improves security—a rogue program cannot extend an oligomorphic class and thus its objects cannot pose as instances of the "innocent" base class. Later we will show that restricting hierarchies to packages encourages the software developers to turn packages into "almost components", leading to better programming style and helping avoid such maintenance hazards as the "fragile base class problem" [10].

To achieve class sealing one can introduce `sealed` modifier to classes [6]. For instance, we can declare a class to be sealed as follows:

```
public sealed class Foo {...}
```

[2] Alternatively, a whole JAR file can be sealed, simply by making the *sealed* attribute at the archive-level.

[3] Note that a signed JAR file cannot be modified.

An alternative to achieve class sealing without language modification is to use the same mechanism as is used in package sealing:

```
Manifest-Version: 1.0
Name: com/ibm/Vehicle/Vehicle.class
Sealed: true
```

Unlike package and JAR sealing we cannot unseal a sealed class, and so we do not have an explicit `false` attribute for class sealing. Once a class is sealed all of its subclasses are also sealed (and we cannot unseal individual subclasses). The main reason for this restriction is that if we allow the unsealing of a sub-class of a sealed class, then the sealed class may cease to be oligomorphic. It is important to note that the concept of sealed classes and sealed packages are orthogonal. Unlike package sealing which inherits the sealing attribute from the JAR sealing attribute, sealed classes do not inherit sealing attributes from their packages. For convenience we can use the wild card `*` for package-level sealing of all classes in a package.

3 Encapsulation

In C we had to code our own bugs. In C++ (and Java) we inherit them.

—Anonymous

Encapsulation is a technique for minimizing interdependencies among separately-written modules by defining strict external interfaces [12]. The three main objectives for enforcing encapsulation are:

1. **Software maintenance:** Modules (or packages) maintain a contract with the external world by restricting the set of methods and fields that can be accessed by the external clients. During module maintenance, a programmer is either forced to leave the module contract intact, or modify it. Modifying the contract of a module may break existing code that relies on the original contract of the module. By enforcing encapsulation we can minimize changes in the client classes during software maintenance.

2. **Security:** Encapsulation can prevent certain kinds of security violation.[4]

3. **Program analysis and optimizations:** Since encapsulation minimizes interdependencies among modules, one can perform a more precise program analysis and optimization. The program analysis and optimization is simplified because we know more about a module when it is properly encapsulated. (See the next section for details).

Encapsulation for OO languages is particularly difficult due to inheritance and subclassing. Snyder pointed out some of key effects of inheritance on ecapsulation [12]. Object-oriented languages, such as Java [2], C++ [14] and Owl/Trellis [11], provide a special set of *access modifiers* for class members, typically `private` (allowing access only from within the class), `public` (allowing access from any class), and `protected` (allowing access from subclasses only).[5] In these cases, the contract with the clients consists of all the `public` members and the contract with the subclasses consists of all `public` and `protected` members.

The fragile base class problem is a nasty issue in class-based objected oriented programming [7, 10]. In Smalltalk, for example, a subclass can access all of the variables inherited from the superclass, thus adding them to the class contract. So if there is a change in the base class all the subclasses are also affected. Also, an invocation of a method on an instance of a class may start a long chain of calls between methods of the class and its superclasses, opening ways to subtle bugs in a subclass. Worse still, a modification of the superclass which does not change neither the protected interface nor its semantics, may break the correctness of the subclass. Such fragile-base class problems have been known to be a source of nightmare during software maintenance.

Encapsulation is even more crucial in Java than in many other OO languages due to Java's security focus and features such as dynamic class loading. Package sealing can be used to ensure integrity of individual packages;

[4]See `http://java.sun.com/security/getSigners.html` for an example of a security bug in the JVM, where an encapsulated variable was accidentally exposed by a method.

[5]In Java, there is default package level access where all classes within the package can access the members, and `protected` in Java also allows default package level access.

however, any class outside the package can be hostile. In this case, it would be most natural to “scope” the access modifiers so that the public interface is still globally available, while the protected interface can be accessed by the members of the package.

Various solutions have been proposed to ameliorate the interaction between encapsulation and inheritance. Szyperski even goes to the extent of abolishing inheritance and subtyping (at least in their present form) [7]. However, in many cases such extreme measures, which make programming less natural, may not be necessary. As Snyder remarked, the complications of inheritance could become less important if “the use of inheritance were confined to individual designers or small groups of designers who design families of related classes.” This is precisely what sealed classes do.

Class sealing offers an elegant solution by restricting inheritance to the package and thus allowing a method to be overridden only within the package but accessible from anywhere in the program. This is applicable to any languages with notions of access control and packages, and not just to Java. The programming style where a limited hierarchy resides in the same package may come in quite naturally in many contexts, including implementation of some design patterns such as State and Strategy [8]. In the next section, we will show that apart from improving encapsulation, such style also helps in optimizing the code to reduce the performance penalty.

Class sealing also helps avoid problems at the subclass-superclass interface. Indeed, if the packages are sealed, all the classes in the package originate from the same source, and, most likely, constitute a single maintenance unit (or part of it). Then updates to classes in the same package are maintained and tested together. Restricting inheritance to a single maintenance unit is thus a solution of the fragile-base-class problem. On the other hand, if related classes belong to different maintenance units, even if developed within the same organization, the problems tend to be discovered at later stages and take more time to fix.

We believe that sealed classes are natural concepts in Component-Oriented Programming (COP). Szyperski succinctly states that: *One thing can be stated with certainty: components are for composition.* Nomen est omen. *Composition enables prefabricated ‘things’ to be reused by rearranging them in ever new composites. Beyond this trivial observation, much is unclear.* We believe that a sealed package in which all classes are also sealed (or final) constitutes a software component. We will call such a package a *Strongly Sealed Package* (SSP). Notice that a client cannot subclass any class from a SSP–classes can only be accessed leading to a composition-style of programming (like the one advocated by COM model).[6]

4 Analysis and Optimization of Sealed Packages

One can take advantage of the oligomorphic property of a class to perform “closed-world” analyses and optimizations. This is because if a reference variable is oligomorphic, then we can compute statically all the types of the objects it may point-to during execution. This observation is key to many interprocedural optimizations of object-oriented languages, such devirtualization and inlining. Notice that in Java, without oligomorpshim, it is often difficult to perform interprocedural analysis and optimization because of dynamic class loading. In the rest of this section we highlight some known optimizations that benefit from knowing references to be oligomorphic. We will also discuss the relationship between sealed classes and extant analysis [13]

Devirtualization

Devirtualization is an optimization that converts virtual method calls to unguarded direct calls. It is possible to do this if we can guarantee that the virtual call has exactly one target for all possible program execution. Computing this information for virtual calls in Java is non-trivial due to dynamic class loading. If a class is oligomorphic and the package in which it is defined is sealed, we are guaranteed that the class hierarchy underneath the class will not extend dynamically and so dervirtualization can be applied safely to messages sent to the oligomorphic class.

Conditional Direct Call Optimization

Knowing a reference is oligomorphic helps in translating a virtual call to a set of guarded direct calls. Modern architecture can then take advantage of guarded calls to do aggressive and speculative optimization. Consider the class hierarchy relation shown in Figure 1. Assume that all classes are in a sealed package. Suppose the `drive()`

[6]See `http://www.microsoft.com/com/`

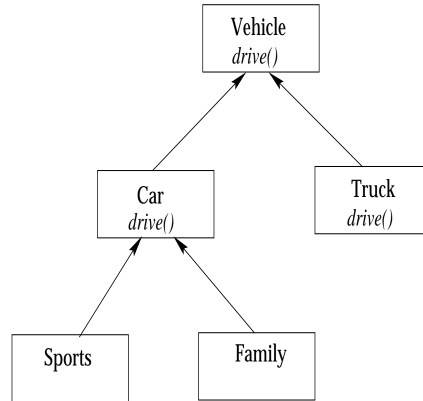

Figure 1: Example subtype hierarchy

method is defined in `Vehicle`, `Car`, and `Truck`. Classes `Sports` and `Family` inherit the `drive()` method from `Car`. Assume that class `Vehicle` is declared to be sealed. This means that the class hierarchy underneath `Vehicle` cannot be extended beyond the sealed package. Now consider a virtual call in a method `foo()` defined in some class:

```
foo(Vehicle veh) {
 ...
 veh.drive() ;
}
```

The target of the virtual call `veh.drive()` depends on the runtime type of `veh`. Most implementations of virtual dispatch load the virtual dispatch table (also called *vtable*), obtain the appropriate offset into the table, and then load the appropriate address of `drive` from the offset and finally invoking method `drive()`. A virtual call requires two loads, and most often the second load is almost always never in the first (and even second) level cache [3].

We propose a simple optimization of virtual calls that requires a load and a branch. Think how one would implement "ad hoc polymorphism" in procedural languages. The first step is to assign a type identifier to each class. Consider the above example, where the declared type of `veh` is the sealed class `Vehicle`. One can encode the class hierarchy rooted at the sealed class `Vehicle` using the following simple type identifiers:

```
enum typeid = { VehicleId, CarId, SportsId, FamilyId, TruckId}
```

We declare a new final field of type `typeid` within each class and assign to it the appropriate type identifier value. Now we generate the following code to replace the call `veh.drive()`:

```
typeid t = veh.typeid ;
if(t == VehicleId  ) {
  Vehicle.drive()
} else if( t== TruckId) {
   Truck.drive()
} else { // (t == CarId || t== SportId || t==FamilyId)
   Car.drive()
}
```

How one encodes type identifiers depends on the object model encoding. At class compilation time we can encode the type identifier and store it as part of the class file (as an attributes). Notice that the above optimization can be performed even if `veh` is not oligomorphic. In this case we also need to introduce an `UnknowId` type identifier, and use the identifier to guard the virtual call `veh.drive()`. But doing the optimization for oligomorphic calls has several advantages. For instance, modern architecture supports features like branch/value prediction, load speculation, etc. If we know that `veh` is oligomorphic then we can inline all of the direct calls, perform aggressive optimization and other speculation/prediction optimization of instructions across methods.

This optimization can be efficiently applied to code implementing several design patterns, such as State or Strategy [8]. In many cases, what in functional languages would be implemented as a switch statement on different functions (doWithStrategyA(), doWithStrategyB(),...), in OO programs is implemented using virtual call (method do() declared in an abstract class Strategy and implemented by classes StrategyA, StrategyB, etc.). While the former way gives better performance and allows more aggressive inter-procedural optimization, the former allows us to produce more flexible and easy-to-maintain code. If all the classes (Strategy, StartegyA, ...) are declared in the same package, class sealing allows us to get the best of both worlds, i.e., writing code with all the advantages of OO and compiling it with all the advantages of non-OO languages.

Extant Analysis

Sreedhar et al. [13] introduced *extant analysis* as a way of performing whole program optimization in the presence of dynamic class loading. The intuition behind that approach is to first define a set of methods and classes as a "closed-world". A simple data flow analysis, called extant analysis, is then performed on the closed-world set to compute whether a reference will definitely point to objects whose type belong to the closed world set or not. If a reference definitely points to objects whose type is in the closed world set, then the reference is called *extant*, otherwise it is called *non-extant* (alternatively, called as *conditionally extant*). One can then perform interprocedural optimization based on extant information. Consider for instance a method-call `p.foo()`. If `p` is extant and there is only one relevant `foo()` in the closed-world, we can devirtualize the call without a guard. But if `p` is non-extant, we have to guard the devirtualized call. Sreedhar et al. also provide simple ways of introducing guards and specializing methods that require guards. Now if we know that `p` is oligomorphic and if the package where the declared type of `p` is defined is in the closed world, then `p` is definitely extant. Unlike Sreedhar et al. we do not need to perform elaborate extant data flow analysis to compute this information.

5 Experimentation

The goal of this section is to explore to which extent class sealing fits into *"current programming practice"* [5]. In particular, we check the relative frequency of the two modes of library classes' usage by applications, namely via instantiation or via subclassing. We also check how many subclassable library classes are subclassed in practice by a representative body of applications.

For our experiments we chose the runtime library (`rt.jar`) distributed with the Sun's Java Development Kit (JDK) version 1.2. The choice was guided by the fact that this library contains multiple and diverse class libraries, such as CORBA, AWT, JDBC, etc. The `rt.jar` library is also special in that it is an ingredient of each and every Java library. For the client applications we chose 45 JAR files of different size, coming from two large IBM applications. One is `jinsight.jar` distributed with Jinsight, a visual tool for optimizing and understanding Java programs.[7] The other 44 come from the distribution of WebSphere Application Server, and include visualization tools such as `console.jar`, XML parsers such as `xerces.jar`, and much more.[8] Together, the benchmarks comprise a behemoth body of code, spanning more than 25 Mb and containing 9332 classes.

Figure 2 shows the distribution of interfaces and classes in `rt.jar`. The `rt.jar` library consists of 4,239 classes and interfaces, comprising 127 packages. Interfaces comprise 9.5% of the library (403 interfaces overall); of the remaining classes, 245 (5.8%) are fully abstract, i.e., define no method implementations, another 262 classes (6.2%) are abstract but do define method implementations; the remaining 3329 classes (78.5% of the total number) are concrete. Out of the 3329 concrete classes, 699 (21.0%) are `final`, and the remaining 3329 classes (78.5% of the total number) are concrete. Out of the 3329 concrete classes, 699 (21.0%) are `final`, and another 1165 classes (35.0%) are identified as oligomorphic due to package sealing and package-level class protection [4]. However, of these 1165 oligomorphic classes, only 54 classes, or 4.6%, have non-trivial hierarchies! This strengthens our assumption that package-level class protection alone would be insufficient to encourage wide usage of oligomorphism.

Our first experiment consisted of analyzing `rt.jar` independent of client applications. We analyzed the class hierarchy graph of `rt.jar` and optimistically marked as *sealed* all classes that were not extended outside their own packages (within `rt.jar`). Of the 3836 classes in `rt.jar` (which includes both abstract and concrete classes) 3256

[7] `http://www.research.ibm.com/jinsight/`

[8] `http://www-4.ibm.com/software/webservers/`

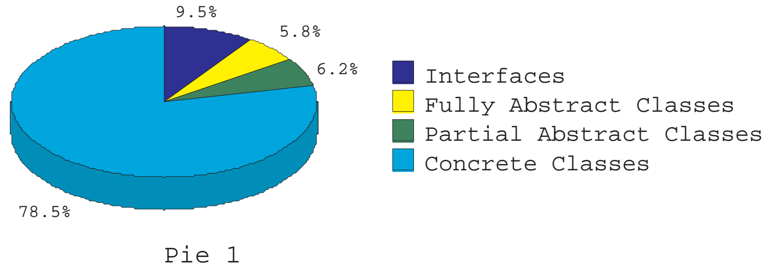

Figure 2: Distribution of Classes/Interfaces in `rt.jar`.

classes (84.9% of the total) were marked as sealed. Of the remaining 580 classes, 291 (50.2%) are abstract, including 51 (8.8%) fully abstract classes.

Our second experiment consisted of analyzing the hierarchy of the `rt.jar` with respect to client applications. As mentioned above our clients include JAR files from WebSphere and Jinsight applications. In the second experiment we determined which of the sealed classes from `rt.jar` were extended by the client applications. Table 1 gives more informations about the 10 largest JARs in the client application set.

	Size (Kbyte)	No of classes	No of fields	No of methods	`rt` packages referenced	`rt` classes referenced
ujc	3,040	2,282	6,450	15,850	19	219
ibmwebas	2,798	1,382	7,862	12,666	18	187
console	2,431	633	3,777	4,628	21	249
ivbjfaceall	989	617	3,013	5,218	21	250
repository	2,165	506	4,570	8,074	9	56
xerces	1,455	512	2,392	5,303	7	74
ejs	666	403	1,576	3,491	13	124
ibmjndi	448	260	1,649	2,592	10	149
dertrjirt	933	240	1,626	1,653	13	189
servletconsole	1,287	286	1,958	2,008	18	158

Table 1: Statistics of the client benchmarks

The next stage was to check the "extension habits" of the client JAR files. We found that the percentage of application classes extending `rt.jar` classes (except for `java.lang.Object`) is generally low, varying between 2.0%-29.0%, with mean percentage 10.5%. Table 2 gives more informations about the types of extension in the 10 largest client JAR files.

	`java.lang.Object`	Same package	Same application	`rt.jar`	`rt.jar` classes un-*sealed*
ujc	53.4%	16.8%	22.1%	7.7%	91
ibmwebas	48.3%	24.3%	19.1%	8.2%	41
console	45.8%	25.1%	4,6%	24.5%	14
ivbjfaceall	31.6%	14.3%	29.0%	25.1%	13
repository	18.6%	29.7%	49.7%	2.0%	0
xerces	39.6%	44.7%	10.2%	5.5%	0
ejs	47.4%	33.4%	11.4%	7.8%	1
ibmjndi	45.2%	31.0%	8.6%	15.2%	0
dertrjirt	44.8%	17.6%	21.1%	16.5%	1
servletconsole	34.3%	33.9%	8.9%	23.0%	5

Table 2: Subclassing behavior of the client JAR files: where do the super-classes come from?

The third experiment was to consult the hierarchy graphs of the client JAR files to see how many `rt.jar` classes tentatively marked as *sealed* were un-*sealed* (i.e., extended) by the client JAR files. We found out that 32 out of the 45 JAR files do not extend *sealed* `rt.jar` classes at all. For example, out of the 10 largest benchmarks in Table 1, three benchmarks, namely `repository.jar, xerces.jar, ibmjndi.jar`, non't un-*seal* any *sealed* `rt` classes at all.

Even libraries that extend *sealed* `rt.jar` classes, do so in a modest proportion. Of the 13 such libraries, 5 extend a single *sealed* class, and only four JAR files (`console.jar, ibmwebas.jar, ivbjfaceall.jar`, and `ujc.jar`) extend more than 10 *sealed* classes in `rt.jar`. For instance, `console.jar`, which references

248 classes from 21 `rt.jar` packages, extends only 14 classes from `rt.jar`, located in 3 packages (1 class from `java.awt`, 1 class from `javax.swing.table`, and the rest from `javax.swing`).

Overall, our results show that only 112 classes in `rt.jar` that were un-*sealed* by all the client JAR files. This still leaves 3143 classes, or 81.9% of all classes, marked as *sealed*. Of course, this is only the rough estimate which cannot replace true design decisions by library developers, but we believe that it constitutes a strong evidence that class sealing fits well into the existing common programming practice. We expect that introduction of sealed classes would lead to changes in coding standards that would make these numbers even higher. Indeed, as we show in Section 3, sealed classes provide an elegant solution to the fragile-base-class problem which should encourage the developers to place the complete implementation hierarchy into a single package.

6 Discussion and Related Work

Java 2 introduced sealing of packages via special declarations in the JAR manifest file. In this paper we extended the notion of sealing to classes. If a class is sealed then we cannot extend the class hierarchy underneath the sealed class beyond the package in which it is defined. Together with sealed packages, sealed classes introduces several benefits including the ability to perform aggressive optimization, security, and software maintenance. Both Sreedhar et al. [13] and Zaks et al. [15] perform analysis to determine sealing information. Both these techniques become very conservative with public classes. What we have done in this paper is to break the semantics of `public` classes into two parts, and when a public class is declared sealed then we are guaranteed that the class hierarchy underneath the sealed class will not extend beyond the package. Also, our change is language neutral, i.e, we declare sealing in the JAR manifest file (although one can introduce a "sealed" modifier to achieve similar semantics [6]). Putting sealing in a policy file such as the manifest file, rather than hard coding it in the language, allows much flexibility.

The notion of oligomorphism was introduced by Biberstein et al. in [4], which discussed the effects of oligomorphism on class members accessibility, object oriented type analysis, and other more complex analyses. The paper also provides statistics on the effect of package sealing on oligomorphism, field accessibility, and inter-procedural analyses on `rt.jar`. The work by Ancona et al. [1] proposes extending Java semantics to support components. In particular, the authors address (and solve) the typing problems that arise if a component extends classes imported from other components. However, the semantic problems that led us to decision that superclasses should not be imported, remain intact.

The notion of package sealing and even class sealing dates back to early language designs. For instance, BETA also has the notion of class sealing and module sealing [9]. The term "sealed" has a different meaning in Beta, in C#, and as defined in the early Dylan language. Sealing in these languages mean "final" (as defined in Java). Subsequently Dylan extended the meaning of sealing by allowing sealed classes to be extended only within a *library*. Dylan uses sealing as a tool to eliminate superfluous flexibility and allow optimizations; sealing is defined there not only for packages, but also for classes and even methods. Also, sealing is hard coded (i.e., a keyword) in the language. The semantics of class sealing in Dylan is that sealed classes cannot be extended outside the declaring *library*, rather than the module. This is good enough to allow the optimizations, but less advisable from the design and reuse points of view. Java packages were designed to be containers for related classes, and since they usually constitute a single maintenance unit, it is advisable to group all the related classes within the package. Also, the "library" approach does not fit the Java model of independent packages which are sometimes JARed together. We feel that our proposal better fits both the current language and the Java philosophy in general.

Sealing does not contradict dynamic class loading: for example, sealed classes can be dynamically loaded to increase flexibility. For example, if the Strategy pattern [8] is implemented with sealed classes to allow optimizations (see Section 4), the specific Strategy can still be chosen at the runtime using the dynamic class loading (`Class.forName("com.ibm.foo.MyStrategy").newInstance()`).

One can seal a particular method instead of the whole class. We can say a method is sealed if (1) the class in which it is defined is sealed, or (2) the user explicitly declares the method to be sealed but not the class in which it is defined. Our semantics of a sealed method is that it cannot be overridden outside the package. A client that is not in the same package can inherit a sealed method or make reference to it. Method sealing have many subtlities such as making sure that a subclass does not override a sealed method. Inner classes introduce more subtlities. Note that unlike sealed classes where we do not even allow subclassing outside a package, an unsealed class containing a sealed method can be subclassed. So clients that sublcass such classes can access protected members of the classes. For now we do not deal with sealing of methods and inner classes.

7 Conclusions

In this paper we demonstrated that class sealing (along with package sealing) has many advantages both in compiler optimizations and in software reuse and maintenance. Class sealing allows finer control on how public classes can be extended. We experimentally demonstrated that class sealing is natural in existing coding practices. Our implementation of class sealing is independent of the core Java language. We merely added new attributes to the JAR manifest file to support class sealing. This allows one to apply class sealing to existing legacy code by appropriately modifying the manifest files.

References

[1] Davide Ancona and Elena Zucca. True modules for java-like languages. In *Proceedings*, Lecture Notes in Computer Science. ECOOP'00, Springer Verlag, June 2001.

[2] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, 2nd edition, 1997.

[3] David Francis Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, Computer Science Division, University of California, Berkeley, December 1997. Report No. UCB/CSD-98-1017.

[4] Marina Biberstein, Yossi Gil, and Sara Porat. Sealing, encapsulation and immutability. In *Proceedings*, Lecture Notes in Computer Science. ECOOP'01, Springer Verlag, June 2001.

[5] Tal Cohen and Jospeh (Yossi) Gil. Self-calibration of metrics of Java methods. In *Proceedings of the 37th International Conference Technology of Object-Oriented Languages and Systems*, pages 94–106, Sydney, Australia, November(20–23) 2000. TOOLS Pacific'00, IEEE.

[6] Apple Computer, Eastern Research and Technology and CA Cuppertino. Dylan — an object-oriented dynamic language, 1992.

[7] Clemens Czyperski. *Component software*. Addison-Wesley, 1999.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[9] O. Lehrmann Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the Beta Programming Language*. Addison-Wesley Publishing Company, 1993.

[10] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In Eric Jul, editor, *ECOOP '98 — Object-Oriented Programming, 12th European Conference , Brussels, Proceedings*, volume 1445, pages 355–382. Springer-Verlag, 1998.

[11] C. Schaffert, T. Cooper, B. Bullis, M. Killian, and C. Wilpolt. An introduction to trellis/owl. In *Proceedings OOPSLA '86*, pages 9–16, 1986.

[12] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings*, pages 38–45. OOPSLA'86, ACM SIGPLAN Notices 21(11), November 1986.

[13] Vugranam C. Sreedhar, Michael Burke, and Jong-Doek Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–207, 2000.

[14] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1991.

[15] Ayal Zaks, Vitaly Feldman, and Nava Aizikovitz. Sealed calls in Java packages. In *Proceedings*. OOPSLA'00, ACM SIGPLAN Notices, November 2000.