"What Programmers Do with Inheritance in Java" - Replicated on Source Code

Ç. Aytekin

cigdemaytekin.872@gmail.com, University of Amsterdam

Abstract. Inheritance is an important mechanism in object-oriented languages and it has been subject of quite some research. Tempero, Yang and Noble made a research [Tempero13] about the usage of the inheritance in Java open source systems and found that the defined inheritance relationships are also used quite considerably in the code, mostly for subtyping and reuse. They also found that late-bound self-reference occurs frequently. They analysed the byte code of the projects. We replicated their study to verify the results by carrying out the same study on the source code. For most of the metrics introduced in their inheritance model we found similar results. We found some suspected false positives in the original study for late-bound self-reference and (external) reuse, but there are not many of them. Except for these cases, our study verifies the correctness of the original study results.

1 Introduction

We externally replicated a study done by Tempero et al. [Tempero13] about inheritance usage. Inheritance is an important mechanism in object-oriented programming. The majority of the studies about inheritance concentrated on the declaration of the inheritance relationships. The original study brought a new perspective on inheritance research. They investigate the *usage* of inheritance in their study, in their own words: "having made the decision to use inheritance at the design level, what benefits follow from the use of inheritance?"

The authors defined an inheritance usage model and analysed the byte code of 93 open source Java projects from Qualitas Corpus, which is a curated collection of open source Java projects [Tempero10]. Their results show that the defined inheritance relationships in projects are frequently used, especially for what they call subtyping and reuse.

Our purpose is to replicate the original study with one major difference: we analyse the source code and not the byte code. We have chosen for replication because of two reasons. Firstly, replication plays a very important role in verification of the results of empirical studies in general. Also in the field of software engineering empirical research this is the case. Brooks et al. explain this in [Brooks08]. Secondly, despite the importance of replication, there are very few replication studies so far, as also shown by Sjoberg et al. in [Sjoberg05]. Replication studies are answers to this need, and so is ours.

Our analysis results are similar to those of original study. But they are not the same. For down-call, we report that 27 % late-bound self-reference (original study reports 34 %) For subtype, we report that at least 61 % of inheritance relations show this usage, whereas original study reports 69 %. Original study reports 22 % of external reuse and 2 % of internal reuse, whereas we report 4 % and 20 % respectively. Our results also show that reuse and subtype explain most of the usages of inheritance and other uses are not significant, just like the original study

We see the following reasons for the differences between the results: the differences between the set-up of two studies, and our limitation about analysing external methods. Our limitation about external method analysis is explained in subsection 5.3. However, we also suspect some false positives in the original study for down-call and external reuse. The reason why we think that there are some false positives is explained in the discussion section (section 8)

This article starts with introducing some of the empirical studies about inheritance. Section 3 contains the definitions of the inheritance model used . In section 4 the original study is explained. Section 5 presents the replication study. The Rascal implementation is explained in section 6. The results of the replication study is presented in section 7. The results are discussed in section 8, followed by the list of threats to the validity of our work (section 9). Finally we conclude with section 10.

2 Related Work

A thorough discussion about the notion of inheritance is given by Taivalsaari in [Taivalsaari96]. Three main usages defined in the original study refer to this article: subtype usage, reuse and down-call (late-bound self-reference). We will define these concepts in detail, but here is a brief explanation in advance. Subtype usage occurs when a child type supplied when parent type is expected, reuse occurs when a child type uses a field or a method defined in its parent and down-call occurs when a method call in a class is directed to a type which is down in the inheritance hierarchy instead of a method in the class itself.

There is a group of empirical studies about inheritance which analyse the code of projects. Tempero and Noble, this time together with H. Melton, carried out the study [Tempero08] using an earlier version of Qualitas Corpus. The study concentrated mainly on how types are defined with respect to inheritance in Java open source projects. In another study, Nasseri, Counsell and Shepperd investigated the evolution of inheritance in Java open source systems (OSS) [Nasseri08]. Lämmel et al. analysed a corpus of .NET projects for the usage of .NET API in the source code [Lammel11]. This last study is also about inheritance usage but from the perspective of API usage.

Another group of empirical studies worked with programmers to observe the effect of inheritance on software quality. An early study of Mancl and Havanas from 1990 [Mancl90], focuses on the effects of using C++ programming language on software maintenance. Similarly, Daly, Brooks, Miller, Roper and Wood also

made an experiment [Daly96] with programmers about the inheritance and investigated if the programs written with inheritance were more easily maintained than the programs written without inheritance. Cartwright replicated the study done by Daly et al., but ended up with opposite results [Cartwright98].

3 Definitions

The authors of the original study proposed a model for inheritance usage. The most important usages of inheritance in their model are subtype, reuse and down-call. In addition to these, they also describe other uses of inheritance, which also occur, but much less frequent than subtype, reuse and down-call.

If a pair of classes has inheritance relationship, it is modelled as an ordered pair of descendant and ascendant. If there is also a usage between the descendant and the ascendant, then this pair is given the attribute which qualifies that certain usage. How often a usage occurs is not taken into consideration here. For example, if a descendant reuses a piece of code from the ascendant, how many times this reuse occurs in the code does not matter.

Here are the definitions of system type, external method, user defined attribute, CC, CI and II attributes, the explicit attribute and indirect reuse. These definitions are important for understanding the scope and the metrics of the original study:

- **System Type** A type (Java class or interface) is a system type if it is defined in the system under investigation. The rest of the types, which are used in the system, but are not defined in the systems are called non-system types. Non-system types are typically defined in the external libraries on which the system under investigation depends on.
- **External method** Similar to non-system types, the methods which are defined outside of the system under investigation are called external methods, or non-system methods.
- **User Defined Attribute:** The descendant ascendant pair in an inheritance relationship has user defined attribute if both of descendant and ascendant are system types. A system type is created for the system under investigation
- **CC, CI and II Attributes:** The descendant-ascendant pair in an inheritance relationship in Java can have one of the three attributes: CC (Class Class), CI (Class Interface) and II (Interface Interface).
- **Explicit Attribute** The inheritance relationship is described directly in the code. Inheritance relation between a child and its direct parent is explicit, whereas a child and its grand parent is not explicit, but implicit.
- **Indirect Reuse** If the inheritance use occurs between the types in a pair which has not explicit attribute, this usage gets the *indirect* attribute. Let us assume that class `GC` extends class `C` and class `C` extends class `P`. If an external reuse occurs between GC and P, all the pairs between GC and P (in this case <GC,C> and <C,P>), are counted as having external reuse attribute (*indirect external reuse*). This is done for the external reuse and subtype

attributes only. For other inheritance usages like down-call or internal reuse, this is not done.

The authors count only explicit user defined pairs in their research.

The inheritance usages we most frequently see in the open source Java projects are subtype, external reuse and internal reuse, which are defined as follows:

- **Subtype:** Subtype inheritance usage happens when a child type is supplied where the parent type is expected. Subtype occurrence can be seen during assignment, casting, parameter passing and returning a parameter in a method declaration in Java. Moreover the enhanced for loop (for each construct) and ternary operator can also contain subtype usage.
- **Internal Reuse:** Reuse occurs if an object of child type accesses a field or calls a method of a parent type. If the reuse happens in the class definition of the child class, this is called internal reuse.
- **External Reuse:** If the reuse happens outside of the class definition of the child class, this is called external reuse.

Down-call is one of the most important concepts of the original study and one research question is solely about down-call. Here is an illustrative example of down-call:

```
class P {
  void p() {
    q();
   }
  void q() {}
}
```

```
class C extends P {
  void q() {}
}
```

In the example, the `p()` method of class `P`, calls method `q()` which is also overridden by its child type. If method `p()` is called on an object of child type, the `q()` method of child type is called instead of the one of parent type. The original study counts the potential down-calls only, in other words, they do not search for a call of method `p()` on an object of child type. In the replication study we did the same, but we find that this approach is open to discussion.

In addition to the most frequent usages, other uses of inheritance are also defined in the inheritance model. Because these usages do not occur frequently, we will only give brief definitions of these concepts. *Category* usage occurs if a parent type has more than one child and at least one of the children have subtype relation with the parent. The siblings which has no other usage with the parent receive category attribute. If an ascendant has only Java constant fields in it, the descendant-ascendant pairs get the *constant* attribute. *Framework* attribute, on the other hand, is given to the descendant-ascendant pairs where ascendant is a child of a non-system type.

The *generic* attribute qualifies a frequently used pattern in raw collections between an ascendant and a descendant. *Marker* usage qualifies the pairs for which ascendants are empty interfaces and descendants implement them for the reason of conceptual classification. Finally, the *super* attribute are given to pairs

in which the descendant explicitly issues a `super()` call to the constructor of the ascending type.

4 The Original Study

4.1 Research Questions

After introducing an inheritance usage model, the original study concentrates on four research questions:

1. To what extent is late-bound self-reference (down-call) relied on in the designs of Java Systems? (the ratio of pairs for which down-call is seen to the total number of class-class inheritance pairs.)
2. To what extent is inheritance used in Java in order to express a subtype relationship that is necessary to design? For class-class pairs, this is the ratio of pairs with subtype usage to the total number of *used* pairs. The *used* pairs are the inheritance pairs for which subtype or reuse is seen. The subtype usage between class-interface and interface-interface pairs are also measured.
3. To what extent can inheritance be replaced by composition? The inheritance relationships which involve internal or external reuse, but which do not involve subtype use, are candidates for such a replacement.
4. What other inheritance idioms are in common use in Java systems? (This is answered by considering the results of various other metrics about other inheritance usages like category, constant, framework, etc.)

4.2 Implementation

The authors developed a Java byte code analysis tool which is based on SOOT framework [ValleeRai99]. With this tool they analysed the byte code distributions of 93 Java projects from the 20101126 release of the Qualitas Corpus [Tempero10]. The study is very well documented in the study web site [Tempero08Web] The article and the study web site have been immensely useful for us when conducting this replication study. When we needed additional information, we e-mailed the authors and we got detailed answers from the first author (E. Tempero). These answers improved our understanding and enabled us to deliver a replication study with better quality.

The original study has some limitations about inheritance model which we also use. The analysis is limited to classes and interfaces, exceptions, enums and annotations are excluded. Moreover only the types which are declared in the project are analysed, and not the third party libraries. Unlike our study, the original study has also some limitations which originate from byte code analysis. In a few cases, for example, source code may not map correctly to byte code.

4.3 Results

For the first question, they conclude that down-call plays a significant role - around a third (median 34 %) of all class-class pairs involve down calls. For the second question, they saw that least two thirds of all inheritance pairs are used as subtypes in the program. About replacing inheritance with composition, the authors found that 22 % or more pairs use external re-use (without subtyping) and 2 % or more use internal re-use (without subtyping or external reuse) which signals opportunities to replace inheritance with composition. For the last question, they report some other uses of Java inheritance (constant, generic, etc.) however the results show that big majority of inheritance pairs (87 %) in their Corpus can already be explained with one of the subtype, external re-use, internal re-use uses and other usages do not occur frequently.

5 Replication Study

5.1 Research Questions

Our research questions directly refer to the four research questions of the original study. For each question we would like to know how our results differ from those of the original study.

5.2 Differences in the Study Set-up

Our study has some differences from the original study in the study set-up. When comparing the results, it is necessary to consider these differences:

- **Source code versus byte code:** The biggest difference between the original and replication studies is about the input to analysis work. The authors analyse the byte code, while we analyse the source code of Java projects.
  Before we started the replication study, we have decided to choose for the source code analysis. There were two reasons for that: firstly, we did not intend to perform an exact replication, we wanted to answer the same research questions from a different perspective, namely by analysing the source code. And secondly, we wanted to use an existing robust tool (or meta-programming language) for analysis. Rascal is proven to be a robust tool for Java source code analysis, but it does not support Java byte code analysis at the moment.
- **Differences between the content of the byte code and the source code:** When we analysed the source code and compared the contents of the source code and byte code distributions, we saw that the set of types which are contained in both distributions differ from each other quite considerably, even for the same versions of the projects.
- **Qualitas Corpus vs. Qualitas.class Corpus:** The authors used the Qualitas Corpus [Tempero10], we also use the Corpus, but the compiled version of it, namely Qualitas.class Corpus [Terra13]. In the original study 93 open

source Java projects are analysed. We could not analyse 3 projects because of errors. From the 90 projects we did analyse, 65 have the same version as the original study and 25 have different versions. The meta-language we used to analyse the source code, Rascal, will analyse the source code correctly when the source code compiles. Therefore it was important for us that the source code compiled correctly. We had two alternatives: we could either use the original corpus and invest time on resolving dependencies of the source code to external libraries and fixing the compilation problems ourselves, or we could use the compiled Corpus (Qualitas.class Corpus), for which this work was already done. We have chosen the second option because of the time limitations, and this meant that we had to analyse different versions of 25 projects.

5.3 Limitations of the Replication Study

The differences between the content of the code analysed is a limitation of our study, as explained in the previous subsection in detail. This difference makes comparison of the results less straightforward.

Another limitation which has impact on our results is about the analysis of non-system methods. We have limited information about external methods. We only analyse the system types, and the external methods are defined in non-system types, i.e. outside of our analysis boundaries. For external reuse, this limitation results in fewer number of external reuse cases for indirect external access (we only mark the child and the immediate parent with external reuse attribute, whereas the original study marks the whole chain between the child and the ascendant which declares the method). For subtype, this limitation affects our analysis during parameter passing. When an external method is called, the parameter types are not totally available for us. We use a very limited heuristic to analyse these calls and it is highly likely that we miss some subtype cases.

There are some more limitations to our study which we think will not have major impact on the results. To name a few: Internal reuse attribute is not analysed for class-interface and interface-interface pairs, for interface-interface pairs category attribute is not analysed, method parameters that are given as ternary operators are not analysed for subtype, etc.

6 Implementation of the Replication Study

We have written a Java source code analysis program in Rascal. Rascal is a meta-programming language which has various features to make (among others) analysis of Java source code easy. Rascal is fully integrated in Eclipse IDE.

With the following simple Rascal code example, we would like to give an idea about how Java source code analysis is done by Rascal:

```
1  public void run() {
2    set[Declaration] pASTs = createAstsFromEclipseProject(|project://cobertura
         -1.9.4.1|, true);
3    pM3 = createM3FromEclipseProject(|project://cobertura-1.9.4.1|);
```

```
4     for (anAST <- pASTs) {
5       visit (anAST) {
6         case m1:\methodCall(_, receiver:_, _, _) : {
7           set[loc] defClassSet = {aClass | <aClass, aMethod> <- pM3@containment
                , isClass(aClass), aMethod == m1@decl};
8           if (!isEmpty(defClassSet)) {
9             loc definingClass = getOneFrom(defClassSet) ;
10            println("The method <m1@decl> is defined in: <definingClass>" );
11            println("Type of receiver is: <receiver@typ>");
12          }
13        }
14      }
15    }
16  }
```

Listing 1.1. Sample Rascal code which analyses a method call

`createAstsFromEclipseProject()` in line 2 is a Rascal method returns all ASTs (Abstract Syntax Trees) for a given Java project. Rascal also creates the M3 model for a given project with method `createM3FromEclipseProject()` as we see in line 3. M3 model contains information about the project from various aspects. This information can be accessed via annotations in Rascal. Some examples of the annotations are: `@extends` annotation (which lists the parent child pairs for classes and interfaces), `@implements` annotation (similar to extends, but for class interface pairs), `@declarations` annotation (which lists the location where different items in the project are declared). In our example, we access to `containment` annotation of project M3 in line 7 to retrieve the class in which the method was declared.

The information in M3 are stored as binary relations (ordered pairs), and Rascal also enables access to binary relations by comprehensions, as we again see in line 7. Once the ASTs are built, it is also possible to visit each node of an AST via the Rascal construct `visit` - line 5 in the example above. The `case` statements (line 6) in the `visit` construct are used for selecting the AST node we are interested in, in this case a `methodCall()`. Once we selected the node we want, it is also possible to retrieve further information about the node itself, like the name of the method that is called (in our example `m1@decl`), if it has a receiver (an object on which the method call was issued - in our case `receiver`) and the type of the receiver (`receiver@typ`).

7 Results

The results of our analysis are, in many cases, similar to the results of the original study, but they are not the same. For most of the inheritance usage we report fewer cases:

For down-call: we observed 27 % (median) of the class-class relations involving potential down-calls whereas the original study reports 34 % median.

For subtype: for class-class pairs, we observed 76 % of subtype usage, just like the original study. Subtype usage can also be seen in class-interface and interface-interface pairs, for class-interface pairs we report a median of 61 % (original study 69 %) and for interface-interface pairs our median is 75 %, while the original median is % 72.

For reuse, we also see that there is opportunity for replacing inheritance with composition. However, we report significantly fewer cases of external reuse, and also significantly more cases of internal reuse. For class-class pairs, our external reuse median is 4 % (original study : 22 %) and our internal reuse median is 20 % (original study 2 %).

For other inheritance cases, we also found some usage, and we also observed that these usages are not significant when compared to subtype and reuse.

Despite not being part of a research question, the perCCUsed (percentage of class-class pairs which have subtype or reuse attribute) is an important metric. For this metric, we have a median of 88 %, while the original study reports 99 %.

The results of the both studies are summarized in table 1.

Inheritance Usage	Replication median (%)	Original median (%).
Down-call	27	34
Subtype - class class pairs	76	76
External reuse - no subtype	4	22
Internal reuse only	20	2
Subtype - class interface pairs	61	69
Subtype - interface interface pairs	75	72

Table 1. Comparative Summary of Results

8 Discussion

The fact that the source code distributions are in many cases very different from the byte code distributions, has a major impact on our results. Moreover, our limitation about the analysis of the external method calls is highly likely to deliver fewer cases in subtype and external reuse. We also did our best to be able to interpret the inheritance model in a sound way, however, there may still be misunderstandings from our side about definitions of certain concepts.

Keeping these limitations in mind, we tried to find out some projects which we could manually investigate to search for reasons for our differences. One small project which had similar byte and source code contents was cobertura (v. 1.9.4.1). For this project, the original study reported five down-call cases, while we could not observe any. Here we include one case for which we suspect a false positive from the byte code analysis. Original study reported a down-call usage between classes GTToken and Token.

The GTToken class from project cobertura has the following source code:

```
1  public static class GTToken extends Token
2  {
3    int realKind = JavaParser15Constants.GT;
4  }
```

An excerpt from the byte code of GTToken is as follows:

```
1     0: aload_0
2     1: invokespecial #10 // Method
3         // net/sourceforge/cobertura/javancss/parser/java15/Token."<init>":()V
4     4: aload_0
5     5: bipush        126
6     7: putfield      #12   // Field realKind:I
7    10: return
```

The definition of down-call makes it necessary that child class GTToken overrides at least one method. In this case, however, we do not see any methods defined by GTToken. GTToken defines only one field. When we look at the byte code, however, we see the command `invokespecial`. We wondered if this was causing the down-call report in the original study. We have e-mailed the authors, and they also could not bring an explanation about this particular case.

From our manual investigation, we also found one more down-call case which is different from the case explained above, for which we also suspect a false positive. For this case, we also mailed the authors and we suspect an interpretation difference for down-call definition between two studies.

We also did a manual investigation for other cases of inheritance usage and we also suspect some false positives for the external reuse, however we did not have enough time to discuss this via e-mail with the authors.

To summarize, we suspect some false positives for down-call and external reuse cases of the original study. We also think that this may be introduced due to the analysis of byte code, in some cases byte code may be misleading. However, because of the limitations of our study, we can not give an exact percentage of false positives, but we do not expect a high percentage of false positives.

9 Threats to Validity

The major threat to validity to our replication is the input we are using. The fact that our input is very different from the original study poses a threat to validity when comparing the results of two studies.

Furthermore, we have a limitation when analysing the external method calls. We know that this causes fewer cases to be counted for subtype and external reuse, but we can not give an exact percentage. This also constitutes a threat for the validity during comparison of subtype and external reuse percentages.

We did our best to understand the inheritance model proposed by the authors. However, there can still be some interpretation differences about the inheritance usage definitions, and this may also pose a threat to our analysis results.

Our minor limitations, which were briefly discussed in section 5.3, will also affect our results and should also be mentioned as, however minor, possible threats to validity.

10 Conclusion

Our conclusions for the research questions are similar to the original study, but they are not the same.

For the first question (about down-call), original study reports about one third of the inheritance relations involving such a case, while we found about one fourth.

For the second research question, we found about 60 % of all inheritance cases involve subtype relationship. The authors also report a higher percentage (66 %) about subtype usage.

About research question three (replacement of inheritance with composition), the pairs without subtype but with reuse attribute are taken into account. Authors found a significant percentage of reuse (median 22 % for external and 2 % for internal reuse). We also report a similar percentage, but the division between internal and external uses is very different in our case (median of 4 % for external reuse and 20 % for internal reuse).

For the last research question, which is about the other uses of inheritance in Java, the authors found out that these occur in many systems, but their use is not generally significant. Although our percentages are not exactly the same with the original study for various other uses of inheritance, our results also agree with this conclusion.

When we investigate the possible reasons for the differences, we see that especially the differences in our study set up play a role here. In addition to this, for the subtype and external reuse, it is highly probable that we report fewer cases because of our limitation of external method analysis. We also conclude that the analysis of byte code can result in false positives in some particular occasions for down-call and external reuse..

As future work for the original study, one can discuss the proposed inheritance model and the metrics and consider some alternatives for detecting and counting various inheritance usages. Especially, how down-call usage is detected and the role of indirect usage in subtype and external reuse, according to us, present some opportunities for further discussion.

Acknowledgments

Author would like to Ewan Tempero for his detailed answers to her questions and to Bas Brekelmans for his help in validating results of this study.

References

[Brooks08] Brooks, A. and Roper, M. and Wood, M. and Daly, J. and Miller, J. Section: "Replication's role in software engineering." from the book : "Guide to advanced empirical software engineering." Springer London, 2008. pp: 365-379.

[Cartwright98] Cartwright, Michelle. "An empirical view of inheritance." Information and Software Technology 40.14 (1998): pp: 795-799.

[Daly96] Daly, J., Brooks, A., Miller, J., Roper, M., & Wood, M. (1996). "Evaluating inheritance depth on the maintainability of object-oriented software." Empirical Software Engineering, 1(2), pp: 109-132.

[Lammel11] Lammel, R., Linke, R., Pek, E., & Varanovich, A. (2011, October). "A framework profile of. net." In 18th Working Conference on Reverse Engineering (WCRE), 2011 (pp. 141-150). IEEE.

[Mancl90] Mancl, Dennis, and William Havanas. "A study of the impact of C++ on software maintenance." In Proceedings of International Conference on Software Maintenance (ICSM), 1990, IEEE, pp: 63 - 69

[Nasseri08] Nasseri, Emal, Steve Counsell, and M. Shepperd. "An empirical study of evolution of inheritance in Java OSS." In 19th Australian Conference on Software Engineering (ASWEC), 2008, IEEE, pp: 269 - 278

[Sjoberg05] Sjberg, D. I., Hannay, J. E., Hansen, O., Kampenes, V. B., Karahasanovic, A., Liborg, N. K., & Rekdal, A. C. (2005). "A survey of controlled experiments in software engineering." IEEE Transactions on Software Engineering, 31(9), pp: 733-753.

[Taivalsaari96] Taivalsaari, Antero. "On the notion of inheritance." ACM Computing Surveys (CSUR) 28.3 (1996): 438-479.

[Tempero08] Tempero, Ewan, James Noble, and Hayden Melton. "How do Java programs use inheritance? An empirical study of inheritance in Java software." In European Conference On Object Oriented Programming (ECOOP) 2008. Springer Berlin Heidelberg, pp: 667-691.

[Tempero08Web] Ewan D. Tempero, Hong Yul Yang, and James Noble. Inheritance Use Data, 2008. Last accessed on 1 September 2014. URL: https://www.cs.auckland.ac.nz/~ ewan/qualitas/studies/inheritance/

[Tempero10] Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M. & Noble, J. (2010, November). "The Qualitas Corpus: A curated collection of Java code for empirical studies." In 17th Asia Pacific Software Engineering Conference (APSEC), 2010, pp. 336-345, IEEE.

[Tempero13] Tempero, Ewan, Hong Yul Yang, and James Noble. "What programmers do with inheritance in Java." European Conference On Object Oriented Programming (ECOOP), 2013 Springer Berlin Heidelberg, 2013. pp: 577-601.

[Terra13] Terra, R., Miranda, L. F., Valente, M. T., & Bigonha, R. S. (2013). "Qualitas. class Corpus: A compiled version of the Qualitas Corpus." ACM SIGSOFT Software Engineering Notes, 38(5), pp: 1-4.

[ValleeRai99] Valle-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., & Sundaresan, V. (1999, November). "Soot-a Java bytecode optimization framework." In Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research (CASCON), 1999, (p. 13). IBM Press.