[Henderson-Sellers, 1996] B.Henderson-Sellers. Object-Oriented Metrics: Measures of Complexity. In: New Jersey, Prentice-Hall, 1996, pp. 142-147.

[Park, 1998] S.Park, E.S.Hong et al. Metrics measuring cohesion and coupling in object-oriented programs. In: Journal of KISS, 1998, 25(12), pp. 1779-87.

[Lorenz, 1994] M.Lorenz, and J.Kidd. Object-Oriented Software Metrics: A Practical Guide. In: New Jersey, Prentice Hall, 1994, p. 73.

[Xenos, 2000] M.Xenos, D.Stavrinoudis, K.Zikouli, and D.Christodoulakis. Object-Oriented Metrics: A Survey. In: Proc. FESMA, Madrid, 2000, pp. 1-10.

Authors' Information

Luis Fernández – UPM, Universidad Politécnica de Madrid; Ctra Valencia km 7, Madrid-28071, España; e-mail: setillo@eui.upm.es

Rosalía Peña – UA, Universidad de Alcalá; Ctra Madrid/Barcelona km 33, Alcalá-28871, España; e-mail: rpr@uah.es

HANDLING THE SUBCLASSING ANOMALY WITH OBJECT TEAMS

Jeff Furlong, Atanas Radenski

***Abstract**: Java software or libraries can evolve via subclassing. Unfortunately, subclassing may not properly support code adaptation when there are dependencies between classes. More precisely, subclassing in collections of related classes may require reimplementation of otherwise valid classes. This problem is defined as the subclassing anomaly, which is an issue when software evolution or code reuse is a goal of the programmer who is using existing classes. Object Teams offers an implicit fix to this problem and is largely compatible with the existing JVMs. In this paper, we evaluate how well Object Teams succeeds in providing a solution for a complex, real world project. Our results indicate that while Object Teams is a suitable solution for simple examples, it does not meet the requirements for large scale projects. The reasons why Object Teams fails in certain usages may prove useful to those who create linguistic modifications in languages or those who seek new methods for code adaptation.*

***Keywords**: Languages; Code reuse; Subclassing*

***ACM Classification Keywords**: D.3.2 [Programming Languages]: Language Classifications – Extensible languages, object-oriented languages; D.3.3 [Programming Languages]: Language Constructs and Features – Classes and objects, data types and structures, inheritance, polymorphism*

1 Introduction

As the requirements for an object-oriented application evolve, so do the applications themselves. For example, an application or even a programming language may need to be enhanced with new functionalities or new linguistic features. Consequently, the set of Java classes that define the application or the set of classes that define a compiler may need to be adequately adapted for such changes [10].

Subclassing is the principle object-oriented programming language feature that provides code adaptation. Patterns may provide an alternative solution, but we are interested in a direct linguistic primitive. Subclassing

allows for the derivation of new classes from existing ones through extension and method overriding. A subclass can inherit variables and methods from a parent class, can extend the parent class with newly declared variables and methods, and can override inherited methods with newly declared ones [10].

When a class that needs to be updated belongs to a collection of classes but is independent from all other classes in the collection, the functionality of that class can be easily updated through subclassing and method overriding. Subclassing is a straightforward code adaptation mechanism in the case of independent classes [10].

However, when working with dependent classes, subclassing may utilize outdated and unintended parent classes, a phenomenon that has been termed the *subclassing anomaly*. Hence, there is a serious concern because the benefits of inheritance are immediately lost in the presence of the subclassing anomaly [10].

Class overriding is a linguistic mechanism to overcome the subclassing anomaly. When new classes are created via subclassing, the correct parent class is overridden and new methods or variables can be introduced. Thus, the subclassing anomaly can be eliminated by using this technique.

Object Teams, an extension to Java, implements a limited version of class overriding and is a proposed method to overcome the subclassing anomaly. It is also an active project today, and is gaining popularity in the Java community. Additionally, Object Teams sums up a good representation of many efforts to enhance Java's code adaptation capabilities with linguistic techniques.

For our evaluation, we use Object Teams version 0.6.1c, which was the latest available release at the time of testing. Our tests were performed during late 2004. Unless otherwise noted, we will indicate Object Teams version 0.6.1c as simply Object Teams.

In this paper, our goal is to evaluate how well Object Teams provides a solution to the subclassing anomaly. We perform this by implementing a complex, real world case study instead of trivial sample code. In section 2, we discuss the subclassing anomaly in more depth and relate it to Object Teams. We present our case study and the evaluation results in section 3. Lastly, in section 4, we conclude our findings, provide future work, and indicate some related works.

2 Subclassing with Object Teams

Applications can be compiled with classes that are either polymorphic or monomorphic references [9]. The subclassing anomaly is limited to occur in monomorphic references. There are three distinct contexts, which include constructor invocations, subclass definitions, and static member accesses. Because the monomorphic reference often points to an old class, new applications must be recompiled to point to an enhanced subclass, which, in turn, may refer to an original parent class. Figure 1 depicts this concept.

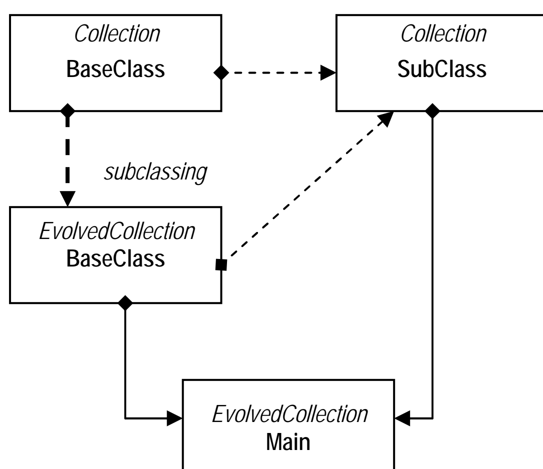

Fig. 1. A subclass definition example.

The outer Java class *Collection* contains *BaseClass*, which is subclassed to *SubClass*. Attempting to reuse code, a programmer subclasses the *Collection* outer class and expects the *Main* class to use newly updated methods in the *Collection.Subclass* class. However, only the methods in *Collection.BaseClass* and *EvolvedCollection.BaseClass* are used. In class overriding, we create a link between the new *BaseClass* and the existing *SubClass*.

There are several options when attempting to overcome the subclassing anomaly. A programmer could reimplement the exact same source code in the new project, if the source is available. This creates obvious code bloating and wastes development time. Alternatively, a project specific fix could be created through the use of additional base classes, interfaces, and message forwarding. However, this option requires some knowledge of how future updates would be implemented, and is often self-defeating. Alternatively, a project could be designed with Object Teams.

Object Teams is an extension of Java [2] with a few new linguistic mechanisms, a new compiler, and an optional new class loader (JMangler). This extension provides much more flexibility without the need to learn a new language because Object Teams is so similar to Java. The new team object extension allows for adaptation and refinement without the expense of new language syntax. Additionally, the compiled class files are native to the JVM, so users need not install a new runtime environment.

In fact, programmers could swap out the Java compiler with the Object Teams compiler. Everything that can be done with the Java compiler can be done in the Object Teams compiler. However, there are three main notable components of the Object Teams language. The first is family polymorphism [2], which redefines the way in which Java handles inheritance. Collections of classes are represented as teams with inner classes. With regard to the subclassing anomaly, some previously problematic inner classes are created as new virtual classes via implicit inheritance. It is this change that allows the subclassing anomaly to be fixed. Object Teams also makes use of translation polymorphism [2], which addresses inheritance with different containment objects. Lastly, many aspect-oriented features are available in Object Teams, such as the ability to remap method parameters and change return types. Such capabilities are not of interest to our goals with the subclassing anomaly.

```
1  public team class Collection {
2    public class BaseClass {
3      public BaseClass() {
4        method();
5      }
6      public void method() {
7        System.out.println("Collection.BaseClass.method()");
8      }
9    }
10   public class SubClass extends BaseClass {
11   }
12 }
13 ______________________________________________________________________________
14 public team class EvolvedCollection extends Collection {
15   public class BaseClass {
16     public void method() {
17       System.out.println("EvolvedCollection.BaseClass.method()");
18     }
19   }
20   public static void main (String args []) {
21     final EvolvedCollection evolvedCollection = EvolvedCollection();
22     evolvedCollection.BaseClass baseClass = evolvedCollection.new BaseClass();
23     evolvedCollection.SubClass subClass = evolvedCollection.new SubClass();
24   }
25 }
```

Fig. 2. A subclass definition: The Object Teams code is very similar to the Java equivalent. For *EvolvedCollection*, the first output is `"EvolvedCollection.BaseClass.method()"` and the second statement correctly outputs the same updated message; although, the Java equivalent would produce the old message. Hence, the subclassing anomaly is indicated and implicitly overcome.

In Figure 2, an example of a subclass definition version of the anomaly is shown. An outer class *Collection* contains inner classes *BaseClass* and *SubClass*. However, a programmer decides to evolve the original

functionality, and wants to change the method within *BaseClass*. In standard Java, the new method is only called when line 22 is executed, not again at line 23. Instead, the original non-overriden method is called. This result is likely to be opposite of the desire of the programmer.

In order to fix this problem while using Object Teams, no additional design strategies are needed. Users can simply develop code as is normally done in Java. The cure is handled implicitly by the Object Teams compiler, specifically by creating virtual classes where they are needed. Minor syntactic changes, such as the keyword `team` denote slight differences with Object Teams code; but, there are no great differences with Java equivalent code. What is different is the presence of the anomaly in the standard Java code and the absence of the anomaly in the Object Teams code.

3 Case Study

The context of the subclassing anomaly is not likely to occur in every updated Java design. However, when software evolution and code reuse are motivations of programmers, problems may quickly be realized. When evolving projects with a simple GUI or theme enhancement, subclassing could be used to change what colors methods paint to a display. More complicated projects may not be linear, and several classes may need to be updated in part so that they are compatible with other original classes. As applications evolve, so do languages and hence compilers.

Our real world test is a case study of an evolving compiler. Because a compiler exhibits a complex dependency relationship between an AST, parser, analyzer, and code generator, it is a very suitable choice to test Object Teams' capabilities. Further, it has been well studied and understood by many. We have implemented the Triangle (educational/testing) language compiler [12] and runtime environment in Object Teams. We can successfully compile the Java equivalent code, but our goal is to evolve the language in some fashion, such as by adding new data types or capabilities. After performing these updates, we find several obstacles that prevent the subclassing anomaly from being solved.

3.1 Object Teams and the subclassing anomaly

Unlike the above simple example where Object Teams worked well, Object Teams has several weaknesses when applied to realistic usages. When typecasting or using the `instanceof` operator in Object Teams, users will quickly find such features to be unsupported. Additionally, a real world project will have complex relationships, and Object Teams fails to always determine a project level dependency analysis. As an alternative to inner classes, users may wish to use packages. However, in some cases with packages, name clashes may cause errors within the Object Teams compiler. Further, extended subclasses may be limited in usage context, as not every situation has been tested by the Object Teams Developers.

Typecasting and instanceof. Attempting to typecast or use `instanceof` with inner classes will produce obvious problems. In the Triangle compiler code, a *Contextual Analyzer* implements a visitor pattern and visits the parsed abstract syntax tree. The *Contextual Analyzer* has a class *Checker*, which determines the types of expressions, among other things. Figure 3 indicates how typecasting and the `instanceof` keyword might be used. An expression in the AST may be visited and then returned, so it can be typecast to determine the type of the expression, such as `int`, `char`, etc. Since *TypeDenoter* is an inner class of *AST*, we can typecast with the form *myAST.TypeDenoter*. However, in Object Teams, there is no support to typecast with inner classes. Similarly, using the `instanceof` operator is not supported. These features are simply issues that have not been fully developed in Object Teams. For small projects, it is possible to simply abandon inner classes and use outer team classes to typecast. However, this modification dismisses the notion of encapsulation and also produces even larger problems in large projects, including the Triangle compiler. Unfortunately, this means that the subclassing anomaly must be solved with other methods, if possible.

```
public Object visitBinaryExpression(myAST.BinaryExpression ast, Object o) {
  myAST.TypeDenoter e1Type = (myAST.TypeDenoter) ast.E1.visit(this, null);
  myAST.TypeDenoter e2Type = (myAST.TypeDenoter) ast.E2.visit(this, null);
  myAST.Declaration binding = (myAST.Declaration) ast.O.visit(this, null);
  if (binding instanceof myAST.BinaryOperatorDeclaration)
    myAST.BinaryOperatorDeclaration bbinding =
                                   (myAST.BinaryOperatorDeclaration) binding;
    ...
  return ast.type;
}
```

Fig. 3. Typecasting and instanceof: Attempting to typecast with inner classes fails because it is not currently supported in Object Teams.

Project level dependency analysis. The Object Teams compiler does not support a project wide dependency analysis on all classes. The consequences of this issue can be indicated in sample code. Figure 4 shows a simple example of how instances of an object, team *AST*, may be used between different outer team classes.

```
public team class SyntacticAnalyzer {
   public AbstractSyntaxTrees myAST = new AbstractSyntaxTrees();
   public myAST.BinaryExpression binaryExpAST = myAST.new BinaryExpression();
   ...
}
____________________________________________________________________________
public team class ContextualAnalyzer {
   public AbstractSyntaxTrees myAST = SyntacticAnalzyer.myAST;
   myAST.Expression expressionAST = SyntacticAnalyzer.binaryExpAST;
   ...
}
```

Fig. 4. Externalized Roles: References must be to the same instance of an object. In the code, there is one instance of *AbstractSyntaxTrees*, which is used to create two other objects, regardless of what class they are located in. However, the Object Teams compiler does not always know that one object is shared through classes.

In our object-oriented compiler case study, methods do the work of returning objects and assigning them to appropriate locations. However, Object Teams requires that if a role (inner class) is to be used outside of its immediate team class, then the role that it is being assigned to must not only have the same team type, but also must be the same team instance. Further, the team object (instance of team class) that it is part of must be denoted as `final`. Type safety could not be guaranteed otherwise, especially if a user changes the type of the team. In order for the compiler to know that each *myAST* is of the same instance of *AST*, it must determine the dependency requirements of the whole project. Such a feature is not yet supported in Object Teams, which makes performing such assignments difficult. A programmer can compile dependent projects knowing which classes to compile in successive order. But, when projects have multiple instances of the same object being referenced from several classes, Object Teams fails to understand that each object is really the same instance, despite help from the programmer. A full project dependency analysis is imperative for this use, something that IBM's Jikes compiler [3] supports so that classes can be built autonomously. Without this feature in Object Teams, the only option to fix this issue is to model more inner classes as outer team classes. This workaround creates the same problems mentioned above and is not suitable for large scale projects.

Packages. As alternatives to inner classes, packages may be used to create a program hierarchy. Each package is represented by a team class and each inner class is defined by a separate file. Using this approach, we can successfully compile the Triangle language in an appropriate package design. However, when we attempt to evolve this package to, say an *EvolvedTriangle* package, we notice some problems. Under some circumstances of the constructor invocation version of the subclassing anomaly, there are compile time errors. These issues refer to name clashes, stack overflow errors, and internal bugs under Object Teams. However, the same source

code, converted to the Java equivalent form, does compile without problems on the regular Java compiler. Unfortunately, executing such Java code quickly proves that the subclassing anomaly is an inherent problem. Hence, there is a small gap of what can be ported from Java to Object Teams.

Usage context. In our case study, we found a class that used throwable objects to catch exceptions. The Triangle compiler has a *Parser* class, which reads the input source files and determines if the format is correct. Figure 5 shows a simplified version of the *Parser* class. If we want to override the *parseDeclaration()* method, necessary if we need to add a new type of declaration to the Triangle language, we can do so by the approach followed in *ExtendedSyntacticAnalyzer*. However, the Object Teams compiler does not correctly determine the correct context of *SyntaxError*. Another bug or lack of design implementation prevents this ability to be used in Object Teams. Although it is possible to compile the original collection, any attempt to override it will produce this error, which prevents us from further testing of the subclassing anomaly under Object Teams. If the equivalent code is tested in regular Java, there are no problems with compilation or usage. This issue further hinders our attempts to produce evolved software.

Using only Object Teams, it is possible to convert all of the Triangle Java source code files to team classes. However, inner classes cannot be used often in our case study, because of the described problems. Therefore, outer team classes may be used as replacements. If a programmer knows all of the limitations and necessary adjustments for Object Teams, it is not a massive task to move original Java code to Object Teams code. However, our real goal is to add extensibility to the Triangle compiler, which requires a new collection or package to be created. Otherwise, we have just remodeled an old piece of software without making any changes. We at least need to make a base to allow for easy future updates. Since there is no method to allow for such evolution, either through inner classes or packages, it is not currently possible to overcome the subclassing anomaly in a real world complex project. Not enough of the compiler has been thoroughly tested or developed; otherwise it ought to be possible to overcome the anomaly on a large scale.

```
public team class SyntacticAnalyzer {
  public class Parser {
    public void parseDeclaration() throws SyntaxError { ... }
  }
  public class SyntaxError extends Exception { ... }
}
________________________________________________________________________
public team class ExtendedSyntacticAnalyzer extends SyntacticAnalyzer {
  public class Parser {
    public void parseDeclaration() throws SyntaxError { ... }
  }
}
```

Fig. 5. Extended classes that throw objects:
Unfortunately, this appears to be an untested and unsupported case in Object Teams.

3.2. Performance evaluation

As part of our goal, we can evaluate the overhead in Object Teams in the cases where it succeeds in providing code adaptation. We can run performance tests on the converted version of the Triangle source, now implemented in Object Teams, as wells as smaller evolution tests. Such results will indicate what penalties there are when working with such a tool that claims to abstract implementation details from the programmer. After a series of detailed tests on Java implementations and their equivalent Object Teams implementations, we can indicate exactly this notion. The most important of these is the run time performance. Object Teams uses and creates more class files, so method lookups and class loading require more time. Also, we measured the extra run time requirements of using overridden classes.

Test Case	Java	Object Teams
Runtime for overridden methods	8 % more	11+ % more
Triangle implementation, compiled classes	115	341 (60 % more bytes)
Compile time	Average	128 % more

Table 1. Java vs. Object Teams Performance:
The most important performance result, runtime, indicates that Object Teams overcomes the anomaly at a small price.

As projects grow large, it can be expected that run time will grow. About eleven percent more runtime is required with Object Teams than for Java equivalent code. The number of classes and hence number of files also increases. The only bound on extra file size would be restricted to just over twice the bytes of the original project, since the worst case is overriding all existing code and adding the Object Teams core code. Overridden classes are achieved by adding virtual classes via implicit inheritance. Thus, original source is virtually copied to new extensions, creating larger class files. The amount of time it takes to compile projects using the Object Teams compiler is not of great concern, since programmers can accept extra time once here, not every time a user invokes an overridden class. Our indications report that the extra time required with Object Teams scales with the size of the project; in our real world example, just more than twice the amount of time is necessary. As long as the run time performance is similar with both techniques, the compilation time can be neglected. The most important performance issues indicate that Object Teams is currently a sound choice for the anomaly, unlike reflection, which requires significantly more run time. If Object Teams will be fixed with solutions to the above problems, these performance tests must be reiterated to determine if greater functionality arrives at performance costs.

3 Conclusion

We have indicated the context of the subclassing anomaly and the problems faced when attempting to implement a solution to a real world complex example. Although Object Teams successfully overcomes the subclassing anomaly in small scale, simple examples, it fails on real world cases. Missing essential features, such as typecasting and instanceof, provide insight to the downfall of Object Teams. A project level dependency analysis that fails and package behavior that is unexpected confirm that not all features are supported in Object Teams. As certain usages produce errors, it is clear that there needs to be additional tests performed on Object Teams by the developers. Our implementations may not have pointed out every issue, as we are only reporting on what is relevant to the subclassing anomaly. Other concerns such as cyclic redundancies and any bugs with the aspect-oriented features remain undocumented. However, at the least, we hope that we have assisted the Object Teams developers by pinpointing some ambiguous areas.

It should be noted that the current version of Object Teams, version 0.6.1c at the time of this writing, is not said to be a final release, but a work in progress. Object Teams developers seek to release a new version, which perhaps will address some of these problems. At such a time, the problems detailed in this paper should be reattempted. If the issues can be corrected without additional performance penalties or additional drawbacks, Object Teams stands to be a possible choice for overcoming the subclassing anomaly. However, until such a time when there is a near perfect solution, we remain without a technique to overcome all versions of the subclassing anomaly on any scale.

There are several other techniques that attempt to overcome the subclassing anomaly. Keris [13] is essentially a flavor of Java, and is strikingly similar to Object Teams, but at the least does not support implicit inheritance or class overriding. MultiJava [1] allows for open classes, but may be difficult to implement and lacks community support for such software. Extendibility is addressed in the generic visitor pattern [8], but any results here are affected by the run time degradation because of the usage of reflection. OpenJava [11] again produces results with reflection, and also is influenced by the run time loss; it, too, is inadequately supported. The most exciting

recent addition is Jx, which attempts to do almost exactly what Object Teams does [6]. The developers have also followed a testing phase that includes nearly an identical plan with the evaluation presented in this paper. Unfortunately, the problems with Jx are often exactly the same as with Object Teams, and Jx is currently not available for public use. None of the available frameworks are the perfect solution because of the difficulty in applying them, the amount of time needed to implement evolved software, the unsupported features, or various other reasons. At first glance, Object Teams appears to excel all of these other alternatives, but it is now apparent that it is not perfect for significant projects. If Object Teams did perform as claimed, it would clearly be a suitable choice for the anomaly and for programmers because of its ease of use, nearly identical syntax as Java, small performance penalties, and compatibility with the existing JVMs. Thus, the overall efficiency of Object Teams far exceeds any other option.

Acknowledgements

This material is based in part upon work supported by the National Science Foundation under Grant Number CCF - 0243284. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Special thanks go to Stephan Herrmann, who provided much needed support for Object Teams.

Bibliography

[1] Clifton, C., Leavens, G., Chambers, C., and Millstein, T., 2000. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. OOPSLA'00, Minneapolis, Minnesota, October 2000, ACM Press, New York, 130-145. Available from <http://www.cs.iastate.edu/~cclifton/multijava-data/papers/TR00-06.pdf>

[2] Herrmann, S., 2004. Confinement and Representation Encapsulation in Object Teams. Technical Report June 2004, Technical University Berlin. Available from <http://www.objectteams.org/publications/TR2004-06.pdf>.

[3] Jikes Homepage. <http://www-124.ibm.com/developerworks/oss/jikes/>.

[4] Joy, B., et al. 2000. Java Language Specification, Addison-Wesley Pub Co, New York, New York.

[5] Laffra, C., 2003. Java Bytecode Manipulation with JikesBT.

[6] Nystrom N., S. Chong, and A. Myers. Scalable extensibility via nested inheritance. Proceedings of Object Oriented Programming, Systems, Languages, and Applications, OOPLSA 04, October 24-8, 2004, Vancouver, British Columbia, ACM, 99-115.

[7] Object Teams Language Definition. <http://www.objectteams.org/def/index.html>.

[8] Palsberg J., and Jay, C. B., 1997. The essence of the visitor pattern. Technical Report 05, University of Technology, Sydney, Australia.

[9] Radenski, A., 2004. Anomaly-Free Component Adaptation with Class Overriding, Journal of Systems and Software 71 (2004), 37-48.

[10] Radenski, A., 2003. The Subclassing Anomaly in Compiler Evolution. International Journal on Information Theories and Applications, Institute of Information Theories and Applications, Sofia, Vol. 10, No 4, 2003, 394-399.

[11] Tatsubori, M., 1999. An Extension Mechanism for the Java Language. Master's dissertation, University of Tsukuba, Tsukuba City, Japan. Available from <http://www.csg.is.titech.ac.jp/openjava/papers/mich_thesis99.pdf>.

[12] Watt, D. and Brown, D., 2000. Programming Language Processors in Java: Compilers and Interpreters, Prentice Hall, New York, New York.

[13] Zenger, M., 2002. Evolving software with extensible modules. In: International Workshop on Unanticipated Software Evolution, Málaga, Spain. Available from <http://zenger.gmxhome.de/papers/use02.pdf>.

Authors' Information

Jeff Furlong – Department of Computer Science, Chapman University, 1 University Dr, Orange, CA 92866, USA; furlo101@chapman.edu

Atanas Radenski – Professor of Computer Science, Chapman University, 1 University Dr, Orange, CA 92866, USA ; radenski@computer.org; http://www.chapman.edu/~radenski