THE SUBCLASSING ANOMALY IN COMPILER EVOLUTION

Atanas Radenski

Abstract. Subclassing in collections of related classes may require re-implementation of otherwise valid classes just because they utilize outdated parent classes, a phenomenon that is referred to as the subclassing anomaly. The subclassing anomaly is a serious problem since it can void the benefits of code reuse altogether. This paper offers an analysis of the subclassing anomaly in an evolving object-oriented compiler. The paper also outlines a solution for the subclassing anomaly that is based on alternative code reuse mechanism, named class overriding.

1 Introduction

Object-oriented applications are collections of related classes. For example, a typical compiler incorporates (1) a set of mutually recursive syntax trees and (2) translation operations on such trees; in an object-oriented compiler, such mutually related trees are implemented as mutually related classes.

As the requirements for an object-oriented application evolve, so should do the applications itself. For example, a programming language may need to be enhanced with new linguistic features, or it may need to have existing features modified. Consequently, an object-oriented compiler for such language may need to have some of its classes adequately adapted.

Subclassing is the principal object-oriented programming language feature that provides code adaptation. (Many patterns have evolved as more robust alternatives to straight forward subclassing for adaptation purposes, but in this paper we are interested in a discussion of linguistic primitives.) Subclassing allows 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.

When a class that needs to be updated belongs to a collection of classes but is independent from all other classes from 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.

Unfortunately, subclassing may not properly support code adaptation when there are dependencies between classes. More precisely, subclassing in collections of related classes may require re-implementation of otherwise valid classes just because they utilize outdated parent classes, a phenomenon that has been termed as the *subclassing anomaly* (Radenski 2002). The subclassing anomaly is a serious concern since it can largely invalidate the benefits of inheritance altogether.

The goal of this paper is to offer an analysis of the subclassing anomaly as it appears in an object-oriented compiler (Section 3). This analysis is preceded by an overview of the subclassing anomaly in domain-independent manner (Section 2). The paper outlines a solution to the subclassing anomaly based on an alternative code reuse mechanism called class overriding (Section 4), and concludes with a discussion of related work (Section 5).

2 Overview of the Subclassing Anomaly

Subclassing in a collection of dependent classes may require re-implementation of otherwise valid classes just because they depend on the parent class. The need to re-implement such otherwise valid classes is referred to as the subclassing anomaly. The subclassing anomaly needs to be understood because it may seriously affect code reusability. This section is devoted to a brief overview of the subclassing anomaly. A more detailed analysis of the subclassing anomaly in a problem independent manner is presented in (Radenski 2002).

Depending on the programming language, a collection of classes can be represented as a namespace (in C#), a stateless package (in Java), or as a package with a state (in Ada 95). In this paper we utilize C# as sample language in order to provide clarity of discussion. However, all results presented in the paper can be applied equally well to virtually any compiled object-oriented language.

Let us assume that in a collection of related classes, a *container* class instantiates and utilizes an object of a *constituent* class. Let us also assume that at a later point of the existence of the collection of classes, the

constituent class needs to be adapted to changing requirements, while the container class remains valid, meaning that it still provides relevant functionality and needs no changes.
Subclassing of the constituent produces an evolved constituent subclass of the original constituent class, which is then incorporated in the evolved collection. The problem is that the integrity of the evolved collection is violated, since in the evolved collection the container class still instantiates and utilizes an object of the old parent constituent class, rather than an object of the evolved constituent class. Even though the container class is assumed to provide relevant functionality, it needs to be re-implemented (which is anomaly), so that it creates an object of the evolved constituent class and thus maintains the integrity of the evolved collection.
Classes may depend on each other in various ways. Some dependencies do not cause anomalies, while others do. The so-called monomorphic dependencies, as defined below, trigger the subclassing anomaly.
Object-oriented languages allow two types of references to classes: polymorphic references and monomorphic references. A *polymorphic reference* to a class *C* stands (1) for *C* itself and (2) for all possible subclasses of *C*. A *monomorphic reference* to a class *C* stands for *C* only but not for any subclasses of *C*.
Polymorphic references to a class *C* occur in:

- parameter, variable, and constant declarations, e.g.: void f (C x); C x;
- type tests, e.g.: if (y is C) …; if (y instanceof C) …;
- type casts, e.g.: x = (C) y;

Monomorphic references to a class *C* occur in:

- constructor invocations, e.g.: *x = new C ();*
- static member access, e.g.: *C.staticMethod ();*
- subclass definitions, e.g.: *class C1 : C {…} ; class C1 extends C {…};*

A class *A* depends monomorphically on class *C* if the definition of *A* contains a monomorphic reference to *C*; further on, we skip the word monomorphically and simply say that *A depends on C*. A class *A* depends on *C* when *A* invokes the constructor of *C*, when *A* extends *C*, or when *A* refers to a static member of *C*.
The subclassing anomaly is triggered by monomorphic dependencies within a collection of classes. When the collection evolves, subclasses can be defined in order to adapt the collection to the changing environment. However, no matter how subclassing is applied, a monomorphic reference continues to stand for the outdated base class in the evolved collection. Thus, all classes that contain monomorphic references must be re-implemented, often in textually equivalent form, as members of the evolved collection. Such re-implemented classes must be recompiled so that monomorphic references are bound to up-to-date subclasses. In contrast to monomorphic references, polymorphic references to outdated base classes do not necessarily require re-implementation of the referring classes - because polymorphic references stand not only for the base class (as monomorphic references do), but for all of its subclasses as well.

3 Analysis of the Subclassing Anomaly in an Evolving Object-Oriented Compiler

This section is devoted to an analysis of the subclassing anomaly in an evolving object-oriented compiler. Our goal is to provide a non-trivial example of the subclassing anomaly as defined in the previous section and to reveal various kinds of class references that trigger the anomaly.
This is not an artificially constructed design example: it is derived from a popular book on object-oriented compilers (Watt, 2000). Depending on one's personal perspective, this design might be considered bad or good (we consider it good), but what is more important, is that it is *common* design which exhibits the subclassing anomaly.
The sample compiler has the usual three phases of syntactic analysis, contextual analysis, and code generation. As shown on Fig. 1, the three phases are implemented as *Parser*, *Checker*, and *Encoder* objects. The parser, checker, and encoder take one pass each, communicating via a syntax tree that represents the source program.
Syntax trees are defined as a hierarchical collection of interfaces and classes. On the top of the hierarchy, a *SyntaxTree* interface encapsulates methods common for all abstract syntax trees (such as a visitor method implemented by both the contextual analyzer and the code generator). Any multiple-form non-terminal symbol is represented by a single interface and several classes that implement this interface, one for each form. For example, statements are represented by the *Statement* interface and several classes that implement this interface, such as *WhileStatement*, *IfStatement*, etc.
The recursive-descent *Parser* class consists of a group of methods *parseN*, one for each non-terminal symbol *N*. The task of each *parseN* method is to performs syntactical analysis of a single *N*-form, and build and

return its syntax tree. These parsing methods cooperate to perform syntactical analysis of a complete program. For example, *parseWhileStatement* performs syntactical analysis of a single *WhileStatement,* and creates and returns an instance of a *WhileStatement* syntax tree (Fig. 1).

```
namespace CompilerCollection {
       public class Compiler {
                              public static void compileProgram (...)
                              {      Parser parser = new Parser (...);
                                     Checker checker = new Checker (...);
                                     Encoder encoder = new Encoder (...);
                                     Program syntaxTree = parser.parse (…);
                                     checker.check (syntaxTree);
                                     encoder.encode (syntaxTree);
                              }
       }
       public abstract class SyntaxTree {… }
       public abstract class Statement : SyntaxTree {…}
       public class WhileStatement : Statement
                      { Expression e; Statement s; …}
                      …
       public class Parser {…
                              public Statement parseWhileStatement ()
                              {      … Expression e = parseExpression ();
                                     … Statement s = parseStatement ();
                                     … return new WhileStatement (e, s);
                              }
                              ….
       }
       public class Checker {…}
       public class Encoder {…}
}
```

```
using CompilerCollection;
namespace UpdatedCompiler {
                      public           class           WhileStatement           :
CompilerCollection.WhileStatement
                      { public void display () {…} }
                      public class IfStatement : CompilerCollection.IfStatement
                      { public void display () {…} }
                      …
                      public class Parser {… // identically re-implemented
                              public Statement parseWhileStatement ()
                              {      … Expression e = parseExpression ();
                                     … Statement s = parseStatement ();
                                     … return new WhileStatement (e, s);
                              }
```

Anomaly triggered by constructor invocation. Suppose that a developer needs to enhance all syntax trees classes with a *display* method, thus converting the *CompilerCollection* into an *UpdatedCompiler* (Fig. 1). One approach is to use subclassing in order to extend with a *display* method all original syntax tree classes, such as *WhileStatement*, *IfStatement*, etc.

Unfortunately, subclassing of the syntax tree classes does not affect any other classes form the *CompilerCollection* and all *parseN* methods from the *Parser* class continue to instantiate the old syntax tree classes. For example, the *parseWhileStatement* method form the *Parser* class, as defined in the *CompilerCollection*, instantiates class *WhileStatement* which is also defined in the *CompilerCollection*.

To effectively update the *CompilerCollection* and convert it into an *UpdatedCompiler*, the developer needs to re-implement the otherwise valid *Parser* class. The re-implementation of the *Parser* class is textually identical with the old one and only needs to be encapsulated within the *UpdatedCompiler*.

The necessity to re-implement a valid *Parser* class is triggered by the subclassing of the syntax tree classes, and this phenomenon is an example of the subclassing anomaly. Note that each *parseN* method from the

Parser class instantiates an object of class *N*, where *N* is the syntax tree representation for *N*. These monomorphic dependencies of class *Parser* on classes *N* trigger the inheritance anomaly.

Anomaly triggered by subclass definition. A developer who needs to enhance all syntax trees classes with a *display* method needs to start with their parent class. Technically, the developer should use subclassing in the *UpdatedCompiler* in order to extend the *SyntaxTree* abstract class with a display method. Unfortunately, the *Statement* subclass of the original *SyntaxTree* class is not affected by this subclassing and remains without a *display* method, as originally defined in *CompilerCollection*. Therefore, the developer needs to re-implement in the *UpdatedCompiler* the otherwise valid *Statement* class. The re-implementation of the *Statement* class is textually identical with the old one and only needs to be encapsulated within the *UpdatedCompiler*.

The necessity to re-implement a valid *Statement* class is triggered by the subclassing of the *SyntaxTree* class, and this phenomenon is an example of the subclassing anomaly. Note that the *Statement* class is defined in the *CompilerCollection* as a subclass of *SyntaxTree*. This monomorphic dependency of the *Statement* class on the *SyntaxTree* class triggers the inheritance anomaly.

Anomaly triggered by static member access. A compiler may use classes with static members for various purposes. For example, all token kinds may be specified as static members of a *Token* class and all operation codes can be encapsulated as static members of a *Machine* class. Parser methods need to access static members of the *Token* class, e.g. *Token.While*. Suppose now that a developer of an *UpdatedCompiler* needs to enhance the *Token* class with a new token, such as *Repeat*. One approach is to use subclassing in order to extend *Token* with a *Repeat* static member. Unfortunately, subclassing of the *Token* class does not affect any of the static references to the original *Token* class, as defined in the *CompilerCollection*. All inherited parser methods continue to use the old version of the *Token class*, while new parser methods that are developed in the *UpdatedCompiler* utilize the updated *Token* class. If the developer wants to have the parser utilize the same *Token* class, the developer must re-implement the whole *Parser* class in the *UpdatedCompiler*. The re-implementations of all inherited parser methods are textually identical with the old ones and only need to be encapsulated within the updated *Parser* class.

The necessity to re-implement valid parser methods is triggered by the subclassing of the *Token* class, and this phenomenon is an example of the subclassing anomaly. Note that the parser methods access static members of *the Token* class. This monomorphic dependency of the *Parser* class on the *Token* class triggers the subclassing anomaly.

4 Elimination of the Subclassing Anomaly with Class Overriding

Class overriding, an object-oriented language feature that is complementary to subclassing, can be used to eliminate the subclassing anomaly (Radenski 2002). In contrast to subclassing, class overriding does not create a new and isolated derived class, but rather extends and updates an existing class. Class overriding is not limited to a single class but propagates across a collection of related classes: it updates all classes from the collection that refer to the class being overridden. Thus, class overriding preserves the integrity of a collection of classes by guaranteeing that any update to a class replaces the previous version of the class within the whole collection.

The definition of class overriding is based on the concept of *replication*. Replication consists in embedding a replica of each class from an existing collection of classes (*the replicated collection*) into a newly created collection of classes (*the replicating collection*). In addition to class replicas, the replicating collection can be further extended with newly defined classed or subclasses.

Replication changes class membership: while all original classes are members of the replicated collection, the class replicas become members of the replicating collection. Except for class membership, class replication preserves all other class properties, including names and access levels. In the replicating collection, each class replica is referred to by the same name and incorporates the same public, protected, and private access levels as the original class in the replicated collection.

A class replica can be overridden (meaning replaced) across the entire replicated collection with its own extension. Similarly to a subclass, the overriding class:

- inherits all data and method members of the class replica
- can override some of the inherited methods
- can extend the replica with additional data and method members

The overriding class replaces the class replica across the entire replicated collection, meaning that all classes from the replicated collection are updated to use the overriding class instead of the replica. Technically this is

achieved by *late class binding*: class references are bound to particular class definitions late, at class loading time, rather than early, at compile time. This is in contrast to traditional compiled languages, such as C#, which use late binding only for methods but limits monomorphic class references to early static binding.
C#, and likewise, various other object-oriented languages, can be enhanced to support class overriding. In C#, collections of classes can be represented as namespaces. Therefore, C# is to be extended with namespace replication statements and with class overriding definitions.

```
namespace CompilerCollection {
       public class Compiler { … }
       public abstract class SyntaxTree {… }
       public abstract class Statement : SyntaxTree {…}
       public class WhileStatement : Statement { Expression e; Statement s; …}
                     …
       public class Parser {… }
       public class Checker {…}
       public class Encoder {…}
}
```

```
namespace UpdatedCompiler {
                     replicate CompilerCollection;
                     override public class WhileStatement  { public void display
() {…} }
                     override public class IfStatement { public void display ()
{…} }
                     …
}
```

Figure 2. Elimination of the subclassing anomaly by namespace replication and class overriding.

A C# outline of a compiler that is updated by means of namespace replication and class overriding – thus avoiding the subclassing anomaly - is presented in Fig. 2. Class overriding updates the *WhileStatement* and *IfStatement* across the entire replicated *CompilerCollection.* No re-implementation of valid classes is needed.

5 Conclusions

This extensibility problem (Findler, 1999; Flatt 1999) appears when a recursively defined set of data and related operations are to be extended with new data variants or new operations. A set of recursive data and related operations can be straightforwardly represented as a collection of dependent classes. Thus, compiler extensibility can be viewed as a special case of recursive class extensibility. Although extensibility can be achieved through subclassing, it requires extensive use of type casts and cumbersome adaptation code, a necessity that is referred to as the extensibility problem.
The compiler extensibility problem can be avoided by following design patterns that are targeted specially at extensibility, such as the extensible visitor (Krishnamurthi et al., 1998), the generic visitors (Palsberg and Jay, 1997), and the translator pattern (Kühne, 1997). Using such patterns implies serious penalties. In the case of the extensible visitor and the translator patterns, the penalty is the significant programming effort needed for an extension. In the case of the generic visitors, the penalty is the significant run-time overhead imposed by the utilization of reflectivity.
Several known linguistic techniques can be applied to attack the compiler extensibility problem, as for example the extensible data types with defaults of Zenger and Odersky (2001) and the evolving open classes of Clifton et al. (2000). None of the known language-level mechanisms seems to offers a silver bullet solution for software evolution. Compared to other approaches, class overriding is simpler and easier to use method to eliminate the subclassing anomaly.

Acknowledgement

This research has been partially supported by a National Science Foundation grant, award number 0243284.

References

1. Clifton, C., G. Leavens, C. Chambers, T. Millstein, 2000. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. OOPSLA'00, Minneapolis, Minnesota, October 2000, ACM Press, New York, 130-145. http://www.cs.iastate.edu/~cclifton/multijava/papers/TR00-06.pdf
2. Findler, R., M. Flatt, 1999. Modular Object-Oriented Programming with Units and Mixins. ACM SIGPLAN International Conference on Functional Programming (ICFP '98), 34(1), 94-104. http://www.cs.utah.edu/plt/publications/icfp98-ff/icfp98-ff.pdf
3. Flatt, M., 1999. Programming Languages for Reusable Software Components. PhD thesis, Rice University, Houston, Texas. http://cs-tr.cs.rice.edu/Dienst/UI/2.0/Describe/ncstrl.rice_cs/TR99-345/
4. Krishnamurthi, S., M. Felleisen, D. P. Friedman, 1998. Synthesizing Object-Oriented and Functional Design to Promote Reuse. ECOOP'98, Brussels, Belgium, July 1998, Springer, Berlin, 91-113.
5. Kühne, T., 1997. The Translator Pattern - External Functionality with Homomorphic Mappings. In Ege, R., M. Singh, and B. Meyer (Eds.), The 23rd TOOLS conference USA 1997, 48-62.
6. Palsberg, J., C. B. Jay, 1997. The Essence of the Visitor Pattern. Technical Report 05, University of Technology, Sydney, Australia.
7. Radenski, A., 2002. Anomaly-Free Component Adaptation with Class Overriding, Journal of Systems and Software, Elsevier Science (under print).
8. Zenger, M., M. Odersky, 2001. Extensible Algebraic Datatypes with Defaults. International Conference on Functional Programming, ICFP 2001, Firenze, Italy, September, 2001. http://lamp.epfl.ch/~zenger/papers/icfp01.pdf
9. Watt, D., D. Brown, 2000. Programming Language Processors in Java: Compilers and Interpreters, Prentice Hall, New York, New York.

Author Information

Atanas Radenski - Chapman University, One University Drive, Orange, California 92866, USA
radenski@computer.org; http://www.chapman.edu/~radenski/

FRONTAL SOLUTIONS: AN INFORMATION TECHNOLOGY TRANSFER TO ABSTRACT MATHEMATICS

V. Jotsov

Abstract: The paper introduces a method for dependencies discovery during human-machine interaction. It is based on an analysis of numerical data sets in knowledge-poor environments. The driven procedures are independent and they interact on a competitive principle. The research focuses on seven of them. The application is in Number Theory.

Keywords: knowledge discovery and data mining, modeling, Number Theory.

1. Introduction

The offered research has begun since 1986 after the exploration of some of the early D. Lenat's papers [Lenat 1976, Lenat 1983]. They gave us the conviction, that the information technologies (IT) are suitable for applications in models which are bounded by Number Theory. The newest evolutionary programming (EP) [EAEA 1997, EA 1997, Nordin 1999] research confirms the possibilities for elaborating new formulas. The considered paper follows the line from our papers [Jotsov1 1999, Jotsov2 1999]. Compared with the works of Lenat [Lenat 1983], or with other sources in the references on informatics, the majority of our papers describe the mathematical results, not the method. The paper's scope is *interdisciplinary* and includes many significantly far research areas. To some extent the proposed method is a continuation of the Lenat's ideas and serves the same purposes: elicitation of new knowledge in the integer data processing, derivation of new formulas, and *whenever possible* generation of new mathematical theorems. At the same time it has some points in common with the Narin'yani's, Shvetsov's constraint programming [Narin'yani 2000,Shvetsov 1997]