Anomaly-Free Component Adaptation with Class Overriding

Atanas Radenski

Chapman University

Department of Computer Science, Mathematics, and Physics

One University Drive

Orange, CA 92866, U.S.A.

mailto:radenski@chapman.edu

http://www.chapman.edu/~radenski/

Abstract

Software components can be implemented and distributed as collections of classes, then adapted to the needs of specific applications by means of subclassing. Unfortunately, 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in collections of related classes may require re-implementation of otherwise valid classes just because they utilize outdated parent classes, a phenomenon that is referred to as the subclassing anomaly.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 component-based programming altogether. We propose a code adaptation language mechanism called class overriding that is intended to overcome the subclassing anomaly. Class overriding does not create new and isolated derived classes as subclassing does, but rather extends and updates existing classes across collections of related classes. If adopted in new languages for component-based programming, or in existing compiled languages such as C# and Java, class overriding can help maintain the integrity of evolving collections of related classes and thus enhance software component adaptability.

Under print in the Journal of Systems and Software, Elsevier Science, Vol. 70-1.

While other techniques such as reflection and binary code adaptation can be used to reduce the magnitude of the subclassing anomaly, class overriding has the advantage of being easy-to-use and efficient.

Keywords: software component adaptation, subclassing, class overriding, the C# programming language, the Java programming language, modular languages

1. Introduction

The Need

Software components are pre-manufactured building blocks that can be assembled into various applications, in order to avoid crafting such applications individually from scratch (Szyperski, 1998). Application assemblers can acquire individual components from different vendors and actually combine the many special skills, ideas, and inventions each vendor has to offer. Component vendors cannot be expected to synchronize their work with each other, but they can build on common 'gluing' standards, such as COM and CORBA (Rogerson, 1996; OMG, 1997) that define common calling conventions. Ideally, neither the components would need to be adapted nor any programming would be required to glue the components together (Büchi and Weck, 1998). In practice, the amalgamation of independently developed components and the adaptation of existing components to the needs of particular applications can suffer form various component incompatibility problems, such as the subclassing anomaly (discussed in this paper).

Object-oriented software components are more likely to be complex collections of related classes, rather than individual classes or libraries of independent classes. Subclassing, the principal code adaptation mechanism that is offered by mainstream object-oriented languages, provides effective adaptation and reuse for individual classes. Unfortunately, adaptation of a collection of related classes may not be effectively provided by independent adaptation of its constituent classes. When a class is adapted to new requirements by means of subclassing, other classes from the same collection may continue to utilize the outdated parent class rather than the newly derived class. Even though such classes may be otherwise valid, they may need to be re-implemented in order to maintain the integrity of the entire collection of classes. This subclassing anomaly may void the benefits of component-oriented programming.

Our Approach

We overcome the subclassing anomaly by means of an alternative code adaptation language mechanism that can be applied to a collection of related classes rather than to an isolated class. Such a collection-wide adaptation mechanism can be beneficial, because object-oriented components and applications are not limited to individual classes. In this paper, we propose *class overriding*, a collection-wide adaptation mechanism that extends and updates an existing class across a collection of related classes. Class overriding can be adopted in new or existing compiled languages, such as C# and Java, as a language feature that is complementary to subclassing and that provides an alternative mechanism for code adaptation. With subclassing, one derives a new class from an isolated parent class; with class overriding, one does not create a new class but rather extends and updates an existing class across an entire collection of related classes, such as a C#

namespace or a Java package. Because class overriding updates all references to a class within a collection of classes, it guarantees the integrity of the collection. While reflection, binary code adaptation, and other existing techniques can reduce the magnitude of the subclassing anomaly, class overriding offers the advantages of being easy-to-use and efficient.

Rationale

Both class overriding and subclassing are forms of class-based (implementation) inheritance. We focus on this type of inheritance because it can be supported by compiled languages in an efficient and reliable manner. It is not the goal of this paper to focus on less popular kinds of inheritance, such as interface-based, prototype-based, and actor-based inheritance, nor to target the less efficient category of interpreted languages.

Recent proposals to overcome component adaptation difficulties typically follow two distinct scenarios: a) design a new language for component-oriented programming with component-adaptation features (Fröhlich and Franz, 2001; Sreedhar, 2001), or b) use an existing object-oriented language and enhance it with component adaptation mechanisms (Aldrich and Chambers, 2001; Zenger 2002). We are among those who prefer to take the object-oriented paradigm as given (on the basis of its contribution to the production of quality software) and investigate how to enhance it so that it provides effective component adaptation as well. The transition from an object-oriented language to its component-oriented enhancement would be easier and less frustrating than the transition to a new language designed from scratch. The problem is not so much that of learning a new language as it is of rewriting a 100,000-line program (Almasi and Gotlieb, 1994).

This paper uses C# and Java as sample languages for the analysis of the subclassing anomaly and for the definition of class overriding. We believe that C# and Java are good choices because these languages represent well a variety of compiled object-oriented languages. Besides, C# is a new language that is developed and promoted by Microsoft and is therefore expected to have strong impact on object-oriented software development, while Java is well established and already has such impact. The subclassing anomaly occurs not only in C# and Java, but also in a variety of similar compiled object-oriented languages, such as C++ and Eiffel, and also in more remote modular object-oriented languages, such as Ada 95, Modula 3, and Oberon 2, to mention a few. The reuse potential of such compiled object-oriented languages can be improved with class overriding. While we utilize C# and Java as sample languages in order to provide clarity of discussion, we also address issues in general and language independent terms whenever appropriate.

Following this introduction, section 2 is focused on the subclassing anomaly. The subclassing anomaly is first discussed in general terms and is then illustrated by a GUI component example. The analysis of the subclassing anomaly provides the ground for Section 3, which is dedicated to class overriding, a mechanism for anomaly-free component adaptation. . This section first defines class overriding in language-independent terms, then specifies how class overriding can be adopted in existing object-oriented languages, such as C# and Java. After that, section 3 demonstrates how class overriding can provide a solution for the subclassing anomaly. Section 3 also describes a prototype implementation of class overriding. Section 4 presents an overview of related

work. Finally, section 5 summarizes the contributions of this paper and outlines opportunities for future work.

2. Analysis of the Subclassing Anomaly

Object technology facilitates code adaptation and reuse through implementation inheritance. Subclassing is the de facto standard object-oriented programming language feature that provides code adaptation. 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.

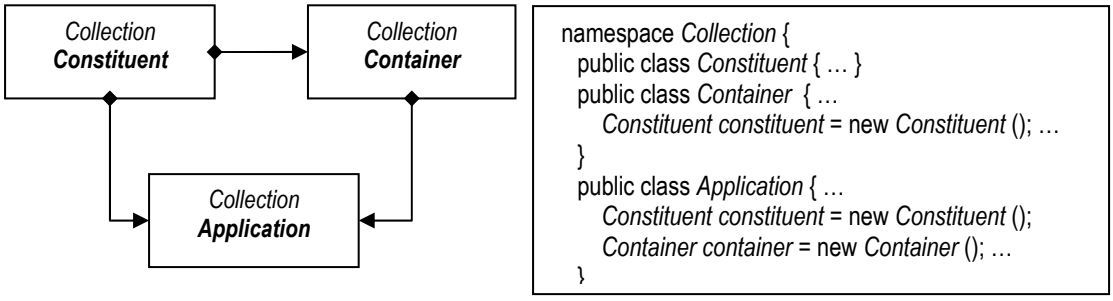
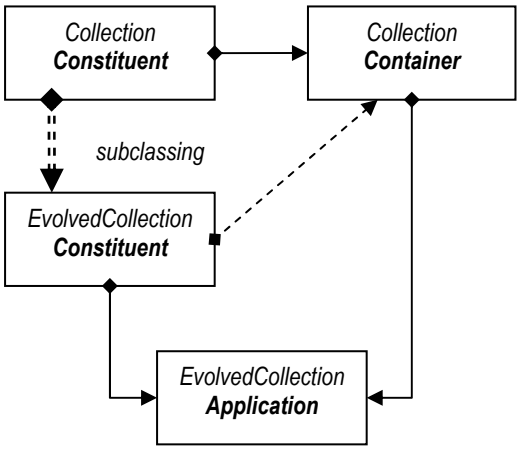


Figure 1. A collection of related classes.
Legend. An arrow from *C* to *A* represents an *A depends-on C* relationship.

An object-oriented component can be implemented as a collection of related classes. In order to adapt the collection to the needs of a particular application the developer may need to update some classes in order to adapt them to specific requirements of that particular application. When a class that needs to be updated is independent from all other classes from the same collection, the functionality of that class can be easily updated through subclassing and method overriding.




```
namespace Collection {
   public class Constituent { … }
   public class Container { …
      Constituent constituent = new Constituent (); …
   }
   public class Application { …
      Constituent constituent = new Constituent ();
      Container container = new Container (); …
   }
}
```

```
namespace EvolvedCollection {
   public class Constituent : Collection.Constituent {…}
   public class Application { …
      Constituent constituent = new Constituent ();
      Container container = new Container (); …
   }
}
```

Figure 2. The subclassing anomaly: The *Collection.Container* class must be re-implemented to instantiate *EvolvedCollection.Constituent*.
Legend. A solid single arrow from *C* to *A* represents an existing *A depends-on C* relationship. A dotted single arrow represents a *depends-on* relationship that is needed to preserve the integrity of the *EvolvedCollection.*

Subclassing is a straightforward code adaptation mechanism in the case of independent classes. Unfortunately, subclassing may not work well for code adaptation when there are dependencies between classes. Let us assume that in a collection of related classes, a container class instantiates and utilizes an object of a constituent class (Fig. 1). 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 (Fig. 2). Even

though the container class is assumed to provide relevant functionality it needs to be re-implemented so that it creates an object of the evolved class and thus maintains the integrity of the evolved collection. In summary, subclassing of constituent classes may require the re-implementation of valid container classes, a phenomenon that we term as the subclassing anomaly.

Classes may depend on each other in various ways. Some dependencies do not cause anomalies, while others do. The so-called monomorphic dependencies that trigger the subclassing anomaly are defined below.

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.

From a language perspective, a collection of related classes is normally packaged in a separate unit, such as a C# namespace, a Java package, an Ada 95 package, an Oberon 2 module, etc. There are certain differences between packaging features that are adopted in various languages; for example, C# namespaces and Java packages do not have state, while Ada 95 packages and Oberon 2 modules do. No matter what the differences are, all mainstream packaging mechanisms suffer from the subclassing anomaly. Our class-overriding solution will be adaptable to various packaging mechanisms.

The rest of this section is devoted to a GUI component example that exibits the subclassing anomaly.

Example: The Subclassing Anomaly in a GUI Component

The programming language C# is used as implementation language in this example. A collection of two related GUI classes, such as *Window* and *Dialog*, is packaged as a C# namespace (Fig. 3). A *Window* object has a title and border and can be used to display information to the user. The *Window* constructor invokes a *PaintGray* method to fill the background of the window in gray. A *Dialog* object is a *Window* that takes input from the user. A *Dialog* object creates a *Window* object then enhances it with input functions. An application can use both *Window* and *Dialog* objects, and each *Dialog* object uses a *Window* object.

Suppose that a GUI application that is to utilize the GUI collection needs to adapt the *Window* class to its own requirements. For example, the GUI application may need a new implementation of the *Show* method so that it paints the window background in blue rather than in gray. Such an adaptation can be achieved by subclassing the existing *Window* class and overriding the inherited *Show* method with a new one (see Fig. 3).

The GUI application is assumed to create both *Window* and *Dialog* objects, as illustrated in Fig. 3. Note that the GUI application instantiates its own derived *Window* class, rather than the parent *Window* class from the GUI collection. For this reason, in the GUI application the *Window* constructor invokes the new *Show* method which on its turn paints the *Window* object in blue, as required. Furthermore, the GUI application instantiates the unchanged *Dialog* class as defined in the GUI collection. For this reason, in the GUI application the *Dialog* constructor invokes the old *Show* method which on its turn paints the *Window* object, and therefore the *Dialog* object in gray, rather than in blue, as required by the GUI application.

```
namespace GUIcollection {
        public class Window {
                virtual public void Show () { PaintGray(); …}
                public Window () { Show (); … }
        }
        public class Dialog {
                Window window = new Window ();
                … add dialog to window …
        }
}
```

```
using GUIcollection;
namespace GUIapplication {
        public class Window : GUIcollection.Window  {…
                override public void Show () { PaintBlue(); … }
        }
        public class Application {
                public static void Main () {
                        Window window = new Window ();
                        Dialog dialog = new Dialog ();
                }
        }
}
```

Figure 3. The subclassing anomaly in a GUI example.

With subclassing, the integrity of the application is compromised, because a method that is supposed to be executed in the same way within the same application is actually executed in a different way. Indeed, one invocation of *Show* executes the overriding implementation *GUIapplication.Window.Show*, while another invocation executes the old overridden implementation *GUIcollection.Window.Show*. Technically, the GUI application paints output-only windows in blue and dialog windows in gray.

To ensure the integrity of the GUI application, the developer needs to re-implement the *Dialog* class. The re-implementation of the *Dialog* class is textually identical with the old one and only needs to be encapsulated within the GUI application namespace.

The necessity to re-implement a valid *Dialog* class is an example the subclassing anomaly. Note that class *Dialog* instantiates an object of class *Window* and incorporate the created instance of *Window* as a data member. This monomorphic dependence of class *Dialog* on class *Window* triggers the inheritance anomaly.

3. Technical Approach to Class Overriding

Class Replication and Overriding

Subclassing in a collection of dependent classes may require re-implementation of all classes that depend monomorphically on the parent class. The need to re-implement such otherwise valid classes is referred to as the subclassing anomaly. The subclassing anomaly is a serious concern since it can largely invalidate the benefits of inheritance. We propose to eliminate the subclassing anomaly with class overriding, an object-oriented language feature that is complementary to subclassing. 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.

Depending on the programming language, a collection of classes can be represented as a namespace (in C#), a stateless package (in Java), as a package with a state (in Ada 95), or as a module (in Oberon 2). We first use the language independent

term *collection* to define class overriding, then we demonstrate how this general definition applies to particular languages, such as C# and Java.

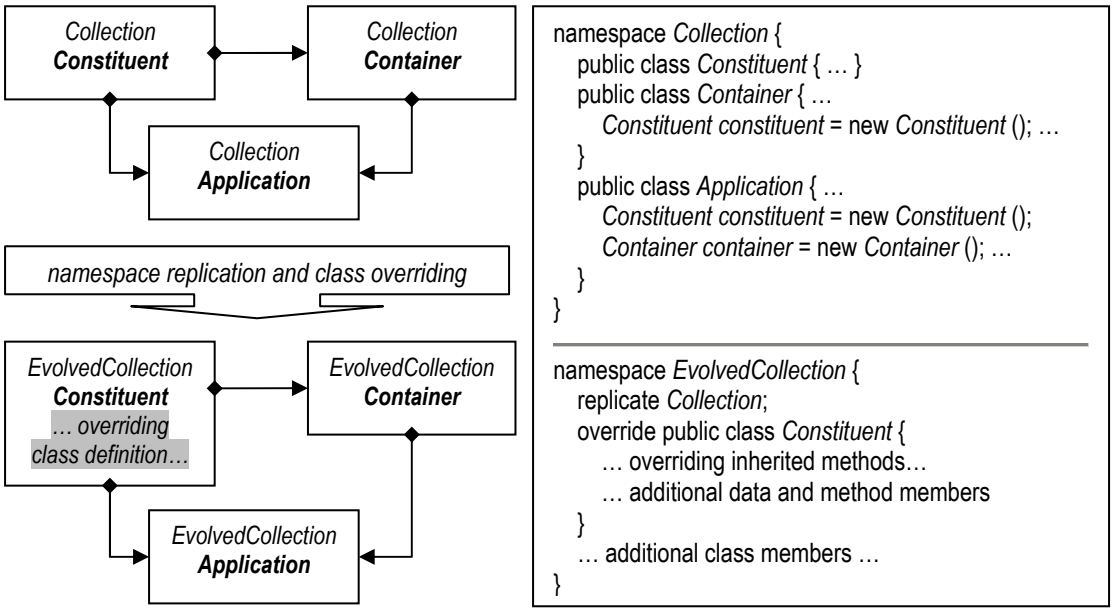


Figure 4. Namespace replication and class overriding in C#.

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# and Java, which use late binding only for methods but restrict monomorphic class references to early static binding.

C# and Java can be enhanced to support class overriding. Collections of classes can be represented as C# namespaces or as Java packages. Therefore, C# is to be extended with a *namespace replication statement*, while Java is to be extended with a *package replication statement*. Furthermore, the two languages are to be extended with *class overriding definitions*. A C# an example of namespace replication and class overriding is presented in Fig. 4. Because Java classes include individual package declaration statements, Java classes should also include individual replication statements. Modular languages such as Ada 95 and Oberon 2 can be likewise enhanced with module replication and class overriding.

Replication versus Import

A programming languages that allows the encapsulation of collections of classes normally supports *collection import* as well. Although import and replication share syntactical similarities, there are very different semantically. The most important differences between replication and import are discussed in this section.

In Java a collection of classes can be encapsulated in a package; classes from the collection can be selectively or entirely imported in any other package. In C# a collection of classes is encapsulated as a namespace; such a collection is *implicitly* imported when it is referenced to. In addition, the C# *using statement* explicitly imports a namespace and assigns an alias to it. Ada 95 offers *with clause* for package import, while Oberon 2 offers an *import declaration* for module import. We ignore some syntactic differences between various language features and use the term *import* in a language independent manner in order to compare import and replication.

The principal difference between import and replication is that import defines *shares-a* relationship between collections of classes while replication defines *contains-a* relationship.

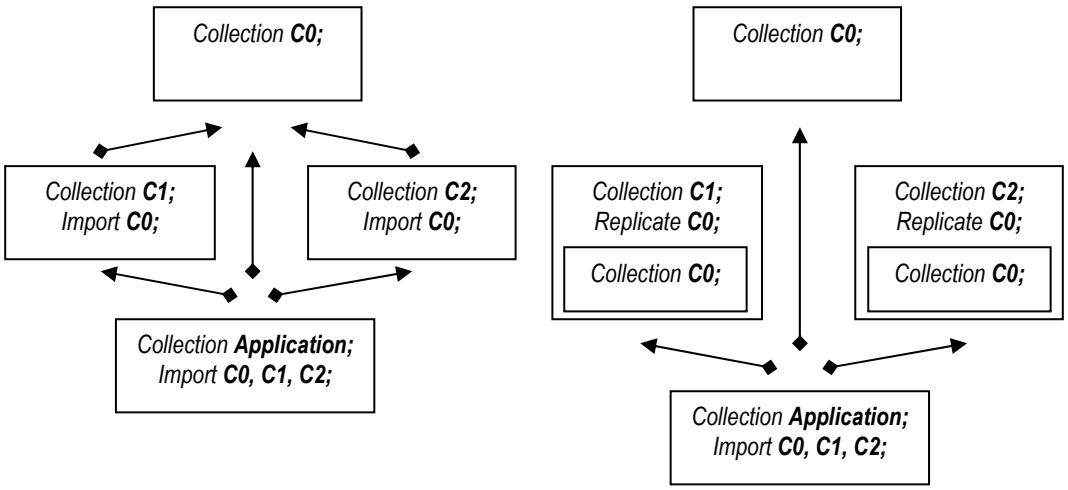


Figure 5. Import (left) versus replication (right).
Legend. *Shares-a* relationship is visualized by arrows; *contains-a* relationship is visualized by nesting.

Consider for example an application in which collections *C1* and *C2* import collection *C0* (Fig. 5, left). The imported collection *C0* is shared between collections *C1* and *C2*. Any change of a static data of a class from *C0* by a class from *C1* is visible for any class from *C2* as well.

Alternatively, assume that collection *C0* is replicated in both collections *C1* and

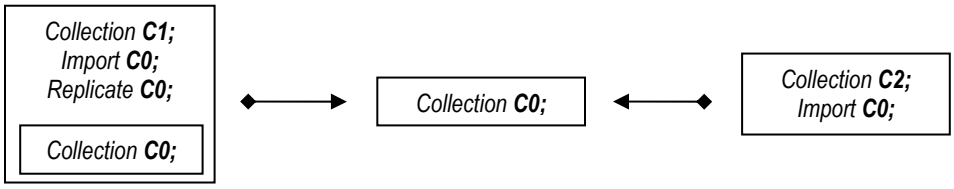


Figure 6. *Shares-a* and *contains-a* relationships can be mixed.

C2. In this case each of the collections *C1* and *C2* incorporates a separate replica of *C0* and therefore has *C0* as its proper part (Fig. 5, right). Thus, *C1* may change static data members replicated from *C0*, but these changes do not affect the same data members

replicated by *C2*. Besides, *C1* and *C2* can define different extensions of the same class from *M0*, simply because they incorporate their own replicas of the class. Likewise, *C1* and *C2* can define different overriding methods for the same method replicated from *C0*.

Note that import and replication can be mixed, if necessary in order to implement *shares-a* and *contains-a* relationships. For example, a collection *C1* can import collection *C0* and, at the same, replicate collection *C0* (Fig. 6). In this case, *C1* embeds a separate replica of collection *C0* and at the same time refers to the shared instance of the imported collection *C0*.

If a replicated class and an imported class have the same name, the replicated class name hides the imported class name. In this case, an unqualified class name resolves to the replicated class, but the imported class can still be referenced through a qualified name.

Consider, for example a GUI collection *A* that defines classes *Window* and *Dialog* (Fig. 7). Consider also a GUI collection *B* that is defined independently from collection *A* and that defines its own classes with the same names, *Window* and *Dialog*, and, in addition, a new *DrawingWindow* class. A GUI application can (1) import collection *B* (by means of a *using statement* in C# in this example), (2) replicate collection *A*, and (3) override the replicated *Window* class (Fig. 7). In the GUI application, the use of the qualified name *GUICollectionB.Dialog* results in a purple window that beeps. In contrast, the unqualified name *Dialog* resolves to *GUICollectionA.Dialog* and therefore delivers a blue window that does not beep. Finally, *DrawingWindow* is defined in collection *B* only and does not need to be qualified.

```
namespace GUIcollectionA {
        public class Window {
                virtual public void Show () { PaintGray (); …}
        }
        public class Dialog {
                Window window = new Window ();
                … add dialog to window …
        }
}
```

```
namespace GUIcollectionB {
        public class Window {
                virtual public void Show () { PaintPurple (); …}
                public Window () { Show (); … }
        }
        public class Dialog {
                Window window = new Window ();
                … sound beep …
                … add dialog to window …
        }
        public class DrawingWindow {
                … unable user to draw …
        }
}
```

```
using GUICollectionB;
namespace GUIapplication {
        replicate GUIcollectionA;

        override public class Window {…
                override public void Show () { PaintBlue (); … }
        }
        public class Application {
                public static void Main () {
                        GUICollectionB.Dialog dialog1 = new GUICollectionB.Dialog ();
                        Dialog dialog2 = new Dialog ();
                        DrawingWindow drawingWindow = new DrawingWindow ();
                }
        }
}
```

Figure 7. Name resolution in a GUI example with replication and import.

Class overriding is possible only within replicated collections of classes, because they are exclusively owned by the replicating collection and not by any other collections. Updating a class requires an exclusive control over the class, as does updating anything else. In contrast to replicated collections, imported collections of classes are shared. Overriding a class from one importing collection may easily generate conflict with other importing collections. For example, two importing collections may try to override the same sheared class. Thus, class overriding can be based on replication but not on import.

Class Overriding as a Solution for the Subclassing Anomaly

The subclassing anomaly, as analyzed in section 2, refers to the necessity to re-implement otherwise valid classes that depend on outdated parent classes in evolving class collections. The subclassing anomaly is triggered by monomorphic references to outdated parent classes. Constructor invocations, subclass definitions, and static member access are all monomorphic references. The problem with monomorphic class references is that they are bound to class definitions statically, at compile time. Classes that contain monomorphic references must be re-implemented, typically in textually equivalent form, as members of an evolved collection of classes. Such re-implemented classes need be recompiled so that monomorphic references are bound to updated subclasses, rather than to their outdated parent classes.

Class overriding, as defined earlier in this section, imposes late binding semantics for all class references. In a language with class overriding, monomorphic class references are bound to particular class definitions at class loading time. This is in contrast to traditional compiled languages, such as C# and Java, which limit monomorphic class references to early static binding.

Class overriding replaces a 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. In the replicated collection, monomorphic references stand for the overriding class, rather than for the parent class. Therefore, valid classes do not need to be re-implemented just because they contain monomorphic references. Thus, the subclassing anomaly is avoided if classes are adapted by means of class overriding, instead of subclassing.

Section 2 introduces GUI component that suffers from the subclassing anomaly. It is possible to modify the GUI component so that it utilizes class overriding instead of subclassing and thus eliminate the subclassing anomaly.

Example: Eliminating the subclassing Anomaly in a GUI

Let us return to the GUI component example from Section 2 which reveals the subclassing anomaly in an evolving collection of GUI classes. As shown in Fig. 3, the collection of GUI classes includes classes *Dialog* and *Window*, such that class *Dialog* depends on class *Window*. Assume again, as in the Section 2 example that a GUI application needs to update the *Window* class with a new version of the *Show* method.

```
namespace GUIcollection {
        public class Window {
                virtual public void Show () { PaintGray(); …}
                public Window () { Show (); … }
        }
        public class Dialog {
                Window window = new Window ();
                … add dialog to window …
        }
}
```

```
namespace GUIapplication {
        replicate GUIcollection;
        override public class Window {…
                override public void Show () { PaintBlue(); … }
        }
        public class Application {
                public static void Main () {
                        Window window = new Window ();
                        Dialog dialog = new Dialog ();
                }
        }
}
```

Figure 8. Elimination of the subclassing anomaly in a GUI example.

The GUI application can replicate the GUI collection and then override the replicated *Window* class (Fig. 8). The overriding *Window* class extends the replicated *Window* class and overrides the replicated *Show* method. Class overriding updates the *Window* class across the entire replicated GUI collection, including the replicated *Dialog* class. For this reason, in the GUI application the replicated *Dialog* class instantiates the overriding *Window* class which supplies the overriding *Show* method. When subclassing is used rather than class overriding, the *Dialog* class needs to be re-implemented because it instantiates the parent *Window* class which invokes the old *Show* method, as demonstrated in the section 2 example.

Implementation

In this paper, we analyze the subclassing anomaly, a software reuse problem that appears in evolving collections of related classes. As a solution to the subclassing anomaly, we propose software reuse mechanism that is based on class replication and class overriding instead of class import and subclassing. In earlier research, we have defined class overriding as an element of a modular language reuse mechanism named *module embedding* (Radenski, 1998). Class replication, as defined in this paper, is the object-oriented alternative to module embedding. We have incorporated class overriding and module embedding in a modular message-parallel language called *Paradigm/SP*. Paradigm/SP has been used to specify and validate generic message-parallel algorithms and to derive various cluster-computing applications from such generic algorithms (Radenski and Norris, 2000).

An implementation of Paradigm/SP has been developed and documented (Radenski, 2000). The implementation of module embedding is based on run-time module, type, and procedure descriptors that are set by a dynamic loading process. Those descriptors bind types and methods to particular definitions at loading time. This implementation technique can be adapted to support class replication and class overriding in non-modular object-oriented languages, such as C# and Java.

Run-type descriptors introduce an additional level of indirection for object references and therefore impose additional run-time overhead. This run-time overhead can be replaced by loading-time overhead if dynamic loading is replaced by dynamic compilation. The development for dynamic compilation techniques for class replication and overriding is a subject for future research.

4. Related Work

A number of researchers have studied and proposed solutions to various component incompatibility problems, such as: component integration and evolution problems, type compatibility problems, conflicts between independent interfaces, the extensibility problem, the fragile base class problem, and the fragile subclassing problem. We agree with researchers who recognize extensibility as the central feature of component-oriented languages (Fröhlich and Franz, 2001), who state that modular (i.e., collection-wise) reasoning is a key requirement for component-oriented programming (Büchi and Weck, 2000), and who claim that components are becoming central to the design process and deserve close integration with the programming language (McDirmid et al., 2001a; McDirmid et al., 2001b). While subclassing and polymorphism support the construction of individual extensible classes, they fail to integrate extensions in collections of related classes. In contrast, class overriding applies to entire collections of related classes and directly enables the *process* of developing and integrating extensions within such collections.

Component integration and evolution problems. Keller and Holzle (1998) assume that a component is a single class and review the so-called component integration and evolution problems. The *component integration problem* appears when an application needs to employ uniformly components from different vendors that have the same functionality but use different method names and signatures. The *component evolution problem* is related to component modifications (such as interface evolution) that may invalidate existing applications based on such components. These problems can be solved by means of a technique called *binary code adaptation* that essentially consists

in byte-code rewriting; this does not require access to source files but is governed instead by rewriting rules that are formalized in special delta files. Binary code adaptation makes systems harder to understand as delta files must also be taken into account. An advantage of class overriding to binary code adaptation is that class overriding is integrated within the programming language and thus does not involve the superfluous language formalisms that are required for binary component adaptation.

Type compatibility problems. Büchi and Weck (1998) argue that evolving software components will need to comply with new contracts during component evolution. Unfortunately, existing components may fail to conform to such new contracts because of limitations of the type system of the underlying programming language. As a technical example, assume that contracts are defined as Java interfaces and consider a class *C* that implements both interfaces *I* and *J*. Assume now that a new contract *K* is defined as a simple extension of the two interfaces *I* and *J*, without the addition of any new features. Despite of the fact that class *C* implements all methods of *K* (because it implements *I* and *J*), formally *C* does not implement interface *K*, due to the name type compatibility rules of Java. Therefore, Büchi and Weck proposed to solve such compatibility problems through the introduction of compound types with structural type equivalency. Alternatively, class overriding can solve such compatibility problems by overriding class *C* and declaring the overriding version as one that implements *K*. An advantage of class overriding is that it avoids the increased complexity of a language that combines both name and structural type equivalency.

Büchi and Weck (1998) describe a hypothetical compatibility problem that involves three or more types, one of which is a class. They have demonstrated that

compound types solve these type compatibility problems as well. Note that the later compatibility problem can be easily resolved through the well-established mechanism of multiple inheritance, or alternatively, through multiple class overriding. Discussion of multiple class overriding is however beyond the scope of this paper.

Conflicts between independent interfaces. Fröhlich and Franz (2000) discuss interface conflicts that appear when a new component tries to implement several existing independent interfaces. For example, two interfaces are in conflict if they specify distinct methods with identical signatures. *Stand-alone messages* are proposed as a solution to *syntactical and semantic conflicts between independent interfaces*. Technically, stand-alone messages are method signatures that are encapsulated in modules and independently of classes and interfaces. New interface modules can be derived from existing ones, and classes can implement interface modules. A programming language with stand-alone messages needs to incorporate both classes and modules, while class replication and class overriding as proposed in this paper do not require modules to belong to the underlying language.

The Extensibility Problem. This problem appears when a recursively defined set of data and related operations are to be extended with new data variants or new operations (Findler, 1999; Flatt 1999). A typical example is an object-oriented programming language translator that as a standard incorporates a set of mutually recursive syntax trees and translation operations on such trees. Should the language be extended with additional features, new syntax trees and operations on them that may need to be added to the existing translator? Although extensibility of a language translator 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*. Class overriding offers an easy to use alleviation to the extensibility problem because it replaces existing classes with their extensions (rather than create new subclasses) and therefore eliminates the need for type casts. Alternatively, Zenger and Odersky (2001a) propose to define default behavior for operations on recursive data sets and argue that inherited implementation of default behavior can serve future extensions if specific adaptations of operations for new data variants are not needed. This approach is materialized as a design pattern for extensible visitors with default cases. While it is possible to encode the extensible visitor pattern in an object-oriented language without relying on additional language features, it is complicated to implement the extensible visitor pattern by hand (Zenger and Odersky, 2001a). Nevertheless, it is worth mentioning that the extensible visitor pattern has been used to design and implement an extensible java compiler (Zenger and Odersky, 2001b), that we plan to use for future implementation of class overriding as an extension of Java.

MultiJava is a backward extension to Java that supports evolving open classes (Clifton et al., 2000). New methods can be added to open classes without creating distinct subclasses and without editing existing code. With open classes, a data type is represented by an abstract superclass and type variants are represented by concrete subclasses. The default behavior is defined as a method for the abstract superclass. If a specific behavior for a variant has to be provided, this method has to be overridden for the variant (Zenger and Odersky, 2001a). In practice, open classes are difficult to use. Whereas a new operation is typically defined as an external top-level method in a single compilation unit, extending or modifying an existing operation can only be done by

explicitly subclassing all affected variants and overriding the corresponding methods. This leads to an inconsistent distribution of code, making it very difficult to group related operations and to separate unrelated ones. Furthermore, extending or modifying an operation always entails extensions of the data type. This restricts and complicates reuse (Zenger and Odersky, 2001a).

Class overriding can add new methods to existing classes exactly as the open class feature of MultiJava does. In addition to adding new methods, class overriding allows programmers to override existing methods without creating distinct subclasses and without editing existing code.

The 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.

A number of object-oriented languages support reflection, a feature that allows programs to examine the structure of classes or objects, and even to change classes (Guimarães, 1998). The main advantage of class overriding to reflection is that class overriding applies to entire collections of classes while reflection normally applies to individual classes. In contrast to class overriding, reflection imposes considerable performance overhead and is relatively difficult to learn.

Aspect-oriented programming (Kiczalez et al., 1997; Elrad et al., 2001) also allows programmers to add methods to existing classes but requires source code for all classes that are extended, while class overriding does not. An alternative approach that does not require source code is to describe class relationships in the Universal Modeling Language (UML), then use reflection to interpret these relationship at run-time, rather than establish them at design time (Lieberherr et al., 2001; Blake and Bose, 2000). While UML and reflection decrease dependencies within class collections, they incur noticeable slowdown (Lieberherr et al., 2001). In contrast, class overriding does not require run-time interpretation and can be compiled into efficient code.

The fragile base class problem. Mikhajlov and Sekerinski (1998) study systematically the *fragile base class problem*, a code inheritance difficulty that was initially informally introduced while discussing object-oriented component standards. The fragile base class problem appears in components that are delivered to users as collections of classes. Users may develop extensions of component classes in order to enhance the functionality of delivered components. Meanwhile, component developers who are unaware of extensions developed by users may produce seemingly acceptable revisions of base component classes that actually invalidate extensions produced by users. Mikhajlov and Sekerinski (1998) formally express five different problematic points of code inheritance that lead to the fragile base class problem. These authors then suggest four restrictions on inheritance that are proven to be sufficient to guarantee safe substitution of a base class with its revision in the presence of extension classes. The subclassing anomaly, as formulated in this paper, appears in components with class dependencies. We overcome the subclassing anomaly by means of class overriding,

which is basically a linguistic mechanism to revise base component classes. Because class overriding is a class revision mechanism, it may potentially generate occurrences of the fragile base class problem. Inheritance restrictions similar to those in (Mikhajlov and Sekerinski, 1998) can guarantee safe class overriding, i.e., class overriding free of the fragile base class problem.

The fragile subclassing problem. Ruby and Leavens (2000) explore the *fragile subclassing problem*, a code inheritance difficulty that is caused by downcalls. A *downcall* occurs in a subclass when a superclass method calls a method that is overridden in the subclass. Downcalls may be problematic because the overriding subclass method may behave differently from what the superclass method expects. Ruby and Leavens propose to eliminate the fragile subclassing problem through (1) some new forms of base class specifications and (2) a set of subclassing restrictions that guarantee that subclasses are free from downcall problems. In very much the same way as subclassing, class overriding can be compromised by improper downcalls. Therefore, the solution to the fragile subclassing problem proposed by Ruby and Leavens (2000) can be adapted to eliminate downcalls from overriding classes.

5. Conclusion

In this paper, we have defined and analyzed the subclassing anomaly, a software reuse problem that appears during the evolution and adaptation of software components that are represented as collections of related classes. We have shown that adaptation through subclassing may violate the integrity of collection of related classes. From a programming language perspective, the subclassing anomaly is triggered by monomorphic class references that stand for classes themselves but not for any

subclasses. We overcome the subclassing anomaly by means of class overriding, a code adaptation mechanism that extends and updates a class throughout entire collections of related classes. Class overriding requires that class references are resolved with late binding, at class loading time, in contrast to traditional compiled object-oriented languages, in which class references are resolved with early binding, at compile time. We have discussed how class overriding eliminates the subclassing anomaly. We have also discussed our experience with implementation and utilization of class overriding.

Class overriding is a generic anomaly-free component adaptation mechanism that is applicable to various component-oriented languages. If adopted in new languages for component-oriented programming, or in existing object-oriented languages such as Java and C#, class overriding can help maintain the integrity of evolving collections of related classes and thus enhance software component adaptability. The analysis of the subclassing anomaly (presented in this paper) can help reveal and avoid possible pitfalls in the design of future component-oriented programming languages; it can also help foresee and avoid possible pitfalls in the development of future component-oriented programming languages.

Acknowledgement

Thank are due to the anonymous reviewers who helped improve the paper by making valuable suggestions. It should be noted that the case presented in Fig. 7 and its discussion in the text was spawned by one of the reviewers.

References

Almasi, G., A. Gotlieb, 1994. Highly Parallel Computing. 2nd Ed., The Benjamin/Cummings Publishing Co., Inc., Redwood City, California.

Aldrich, J., C. Chambers, 2001. Architectural Reasoning with ArchJava. First OOPSLA Workshop on Language Mechanisms for Programming Software Components, Tampa Bay, Florida, October 2001, 22-31. http://www.cs.iastate.edu/~leavens/SAVCBS/papers-2001/TR.pdf

Blake, B., P. Bose, 2000. An Agent-based Approach to Alleviating Packaging Mismatch. 4th International Conference on Autonomous Agents (AGENTS2000), Barcelona, Spain, June 2000, ACM Press, New York, 64-69. http://cssun.georgetown.edu/~blakeb/pubs/agents2000.pdf

Büchi, M., W. Weck, 2000. Generic Wrappers. ECOOP'00, Cannes, France, June 2000, Springer, Berlin, 201-225. ftp://ftp.abo.fi/pub/cs/papers/mbuechi/ECOOP2000.pdf

Büchi, M., W. Weck, 1998. Compound Types for Java. OOPSLA'98, Vancouver, BC, Canada, October 1998, ACM Press, New York, 362 – 373. ftp://ftp.abo.fi/pub/cs/papers/mbuechi/OOPSLA98.pdf

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

Elrad, T., M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher, 2001. Discussing Aspects of AOP. CACM 44(10), 33-38.

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

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/

Fröhlich, P., M. Franz, 2001. On Certain Basic Properties of Component-Oriented Programming Languages. First OOPSLA Workshop on Language Mechanisms for Programming Software Components, Tampa Bay, Florida, October 2001, ACM Press, New York, 15-18.
http://www.ccs.neu.edu/home/lorenz/oopsla2001/23_Frohlich.pdf

Fröhlich, P., M. Franz, 2000. Stand-Alone Messages: A Step Towards Component-Oriented Programming Languages. In: Gutkneht, J., W. Wech (Eds.), Modular Programming Languages, Springer, Berlin, 90-103.
http://nil.ics.uci.edu/~phf/pub/tr-ics-2000-18.pdf

Guimarães, J., 1998. Reflection for Statically Typed Languages. ECOOP'98, Brussels, Belgium, July 1998, Springer, Berlin, 440-461.

Keller, R., U. Holzle, 1998. Binary Component Adaptation. In: Jul, E. (Ed.), ECOOP'98 Conference Proceedings, Lecture Notes in Computer Science 1445, Springer, Berlin, 307-329.
http://www.cs.ucsb.edu/labs/oocsb/papers/TRCS97-20.pdf

Kiczalez, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin, 1997. Aspect-Oriented Programming. ECOOP'97, Jyväskylä, Finland, June 1997, Springer, Berlin, 220-242.

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.

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.

Lieberherr, K., D. Orleans, J. Ovlinger, 2001. Aspect-Oriented Programming with Adaptive Methods. CACM 44(10), 39-41.

McDirmid, S., M. Flatt, W. Hsieh, 2001a. Jiazzi: New Age Components for Old Fashioned Java. In the proceedings of OOPSLA 2001, Tampa, Florida, 211-222. http://www.cs.utah.edu/plt/jiazzi/download/jiazzi01oopsla.pdf

McDirmid, S., M. Flatt, and W. Hsieh, 2001b. Mixing COP and OOP. Proc. First OOPSLA Workshop on Language Mechanisms for Programming Software Components, Tampa Bay, Florida, October 2001, ACM Press, New York, 29-32. http://www.ccs.neu.edu/home/lorenz/oopsla2001/31_McDirmid.pdf

Mikhajlov, L., E. Sekerinski, 1998. A Study of the Fragile Base Class Problem. ECOOP'98, Brussels, Belgium, July 1998, Springer, Berlin, 355-382. http://www.cas.mcmaster.ca/~emil/publications/fragile/ecoop98.pdf

OMG (Object Management Group), 1997. The Common Object Request Broker: Architecture and Specification. Revision 2.0, formal document 97-02-25.

http://www.omg.org.

Palsberg, J., C. B. Jay, 1997. The Essence of the Visitor Pattern. Technical Report 05, University of Technology, Sydney, Australia.

Radenski, A., 1998. Module Embedding. International Journal Software - Concepts and Tools 19(3), 122-129. http://www.chapman.edu/~radenski/research/papers/module.pdf

Radenski, A., B. Norris, 2000. Generic Cluster-Computing Algorithms and Applications. International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, June 26-29, 2000, CSREA Press, 485-491.

Radenski, A., 2000. The Paradigm/SP Prototyping and Specification Message-Parallel Language. http://www.chapman.edu/~radenski/research/language.html

Rogerson, D., 1996. Inside COM. Microsoft Press, Redmond, Washington.

Ruby, C., G. Leavens, 2000. Safely Creating Correct Subclasses without Seeing Superclass Code. OOPSLA'00, Minneapolis, USA, October 2000, ACM Press, New York, New York, 208-228.

Sreedhar, V., 2001. ACOEL on CORAL: A Component Requirement and Abstraction Language. First OOPSLA Workshop on Language Mechanisms for Programming Software Components, Tampa Bay, Florida, October 2001, 125-131. http://www.cs.iastate.edu/~leavens/SAVCBS/papers-2001/sreedhar.pdf

Szyperski, C., 1998. Component Software. ACM Press/Addison-Wesley Publishing Co., New York, New York.

Zenger, M., M. Odersky, 2001a. Extensible Algebraic Datatypes with Defaults. International Conference on Functional Programming, ICFP 2001, Firenze, Italy, September, 2001.
http://lamp.epfl.ch/~zenger/papers/icfp01.pdf

Zenger, M., M. Odersky, 2001b. Implementing Extensible Compilers. ECOOP'01 Workshop on Multiparadigm Programming with Object-Oriented Languages, Budapest, June 2001.
http://lamp.epfl.ch/~zenger/papers/mpool01.pdf

Zenger, M., 2002. Type-Safe Prototype-Based Component Evolution, ECOOP'02, Malaga, Spain, June 2002, Springer, Berlin, 470-497.
http://icwww.epfl.ch/publications/documents/IC_TECH_REPORT_200214.pdf

Vitae

Atanas Radenski (http://www.chapman.edu/~radenski) is a Professor of Computer Science in Chapman University, California. His research interests are in the areas of programming languages, object-orientation, parallel computing, distributed computing, and evolutionary computing. He has authored about 70 publications. His research has been supported by grants from the National Science Foundation, from the National Aeronautics and Space Administration, and from other agencies.