3. Summary

Object-oriented programming promises to improve programmer productivity by fostering component reuse. Rather than being rewritten from scratch, new programs are mostly composed of existing components which can be adapted to the specific purpose by using language mechanisms such as polymorphism and inheritance. Thus, relatively little time is spent creating entirely new code, and most of the programming effort lies in combining or specializing existing reusable components [Cox86]. In other words, the main activity associated with software development is not the origination of new programs but the integration, modification, and evolution of existing ones [Win79].

A central idea behind object-oriented programming is that changing existing source code should be avoided: if programmers copy and then edit code, many redundant but slightly different versions of the code will quickly lead to maintenance problems. Instead, existing functionality is customized by creating new subclasses or implementing predefined interfaces in frameworks. Rather than changing existing code, the customization happens by adding code, leaving a single version of the base functionality that is easier to maintain.

Unfortunately, this vision cannot always be realized in environments that cannot be centrally managed and coordinated. In particular, combining independently developed components can be difficult, as can be the evolution of components over time. Even if small (trivial) changes could rectify a problem, source code changes may be impossible for a variety of reasons. Therefore, interoperability between software components is a major issue in software development [Weg96], and many approaches to improve component reuse and integration have been proposed.

Our approach accepts that components are not perfectly coordinated, and instead looks for mechanisms that allow more flexible object couplings. In the remainder of this proposal, we will first discuss the dual problems of integration and evolution using examples taken from Java libraries. We then describe a new technique, *binary component adaptation (BCA)*, and demonstrates how it solves most of the problems. BCA shifts many small but important decisions (e.g., method names or explicit subtype relationships) from component production time to component integration time, thus enabling programmers to adapt even third-party binary components to their needs.

Binary component adaptation rewrites component binaries before or while they are loaded. This rewriting is possible if binaries contain enough symbolic information (as do Java class files, for example). Component adaptation takes place *after* the component has been delivered to the programmer, and the internal structure of a component is directly modified *in place* in order to make changes. Rather than creating new classes such as wrapper classes, the definition of the original class is modified. By directly rewriting binaries, BCA combines the flexibility of source-level changes without incurring its disadvantages. In particular, binary component adaptation

- requires no source code access, so that it can be used on third-party libraries;
- preserves *release-to-release compatibility*, so that compatibility problems only arise in situations where they would also arise with unmodified components;
- is very flexible, allowing a wide range of modifications (including method addition, renaming, and changes to the inheritance or subtyping hierarchy);
- can be deferred until load-time, so that the adaptations can be distributed and performed “just in time”; and
- is implementable with low load-time overhead.

We propose to implement binary component adaptation for Java to show that an efficient implementation is feasible, and to make this implementation available publicly to allow an evaluation of BCA’s usefulness based on the feedback received from external users.

1. Information about Principal Investigator 1

2. Cover Sheet 2

3. Summary 3

4. Proposed Work 5
4.1 Background 5
4.1.1 The Integration Problem 5
4.1.2 The Evolution Problem 6
4.1.3 Summary 7
4.2 Binary Component Adaptation (BCA) 8
4.2.1 Overview 8
4.2.2 Background: JVM class files 9
4.2.3 Delta Files 10
4.2.4 Potential Adaptations in Java 10
4.2.5 Enforcing Binary Compatibility 11
4.2.6 Summary 12
4.3 Discussion 12
4.3.1 Examples revisited 12
4.3.2 Deltas as First-Level Components 13
4.3.3 Distribution of Deltas 13
4.3.4 Limitations 14
4.4 Related Work 14
4.5 Plan of Work 16
4.6 Expected Results and Impact of Proposed Work 17
4.7 Results From Prior NSF Support 18

5. References Cited 20

6. Curriculum Vitae 22

7. Budget 24

8. Current and Pending Support 30

9. Facilities, Equipment, and Other Resources 32

4. Proposed Work

4.1 Background

Component reusers face two main problems, integration and evolution. In this section, we briefly discuss these problems. Even though we use examples taken from Java, the problems per se apply to all object-oriented languages.

4.1.1 The Integration Problem

Assume an application using components A and B, each obtained from a different vendor or organization. The simplified class interfaces of these types are as follows (in reality both classes would define additional methods, of course):

```
class A {
    public void output(PrintStream os);
}

class B {
    public void print();
}
```

Note that both classes define support for printing, although the details (method names and signatures) differ slightly. Unfortunately, these minor differences suffice to make the two classes hard to integrate into the same program. For example, suppose a programmer wanted to store components derived from A and B in a list in the application, and later iterate over the list to print all objects:

```
Enumeration e;
while (e.hasMoreElements()) {
    Printable p = (Printable)e.nextElement();
    p.print();
}
```

However, this usage is impossible because of two superficial problems. First, in class A the print method is called output and expects a stream as a parameter. If the programmer had access to source code, it would be possible to add a print method to A; its implementation would simply call output(System.out). But source code access is unlikely since the component was bought from an independent vendor.

Even after solving the first problem, the two components still cannot be combined because of a second problem: A and B have no common superclass or supertype. This problem could be resolved by adding an implements Printable clause to both classes, indicating that they both support the interface Printable. Again, this change requires source code access and a recompilation.

Alternatively, the component user could try to solve integration problems using wrapper objects or other programming techniques. However, such attempts can introduce significant additional programming effort, runtime overheads, and difficult subtyping problems [Höl93] and thus we do not consider them a satisfactory general solution.

One could argue that integration would be seamless if components were "well-designed" right from the beginning. However, we believe that it is highly implausible that such perfection would be common in a open, large component market.[1] Components cannot be viewed in isolation: even if all components are internally consistent and well-designed, their combination may not be consistent. Programs are likely to combine may different components or component frameworks, making perfect harmony unlikely.

Sometimes, "imperfection" is present by design. For example, while it might be very desirable for some applications that the standard class String contain encrypt and decrypt methods, such functionality may have been considered too specialized by the designers of class String and thus was deliberately not included

[1] A detailed argument can be found in [Höl93].

in order to keep the class interface simple. Other component programmers may well have different or even conflicting requirements and therefore would choose a different interface. Even if a component producer could anticipate all possible uses a priori (and that is a big "if"), the resulting interface would be too complicated, and most programmers would be overwhelmed by a multitude of methods, most of which they never need.

In summary, minor incompatibilities prevent pervasive reuse because a few details do not match: a method is misnamed, parameters may appear in a different order, a method requires a slightly different argument type, a component does not fit into a type hierarchy, etc. Therefore, components must be adapted in some manner before they can be successfully integrated into an application.

4.1.2 The Evolution Problem

Evolution represents the continuous cycle of activities involved in the development, use, and maintenance of software systems. Software systems evolve over time in response to numerous requirements, including bug fixes, user demands for greater functionality, and especially to support changes in related software [Som92]. Because such change is inevitable, mechanisms must be developed for evaluating, controlling and making modifications.

In widely distributed and relatively loosely coordinated environments like the Internet, it is impractical or impossible to automatically recompile pre-existing binaries that depend on a component that is evolving. Thus, component evolution should preserve binary compatibility with pre-existing applications. A change is *binary compatible* with pre-existing client binaries if these binaries that previously executed without linkage errors continue to execute without linkage errors. Linkage errors are type errors detected at link time (e.g., missing method or mismatched method signatures) or at runtime (e.g., invoking an abstract method). Of course, the actual intention of binary compatibility is to ensure that existing binaries continue to run *correctly* with the new component version, but such strong semantic guarantees cannot be verified by a compiler.

Therefore, the evolution of a class library distributed over the Internet is only practical if changes do not abandon support for the already compiled applications. In other words, a library vendor must maintain release-to-release binary compatibility [F+95].

4.1.2.1 Interface Evolution

The requirement for binary compatibility places stringent restrictions even on the use of simple language constructs such as Java's interfaces. For example, assume an interface Enumeration as shown below which provides a uniform way to iterate through various collections:

```
interface Enumeration {
    boolean hasMoreElements();
    Object nextElement();
}
```

The method hasMoreElements checks for the end of the iteration and the method nextElement returns the next element of the collection.

After this interface was released, suppose we would like to extend it with a method that returns the last element:

```
Object lastElement();
```

However, adding a new member to an interface requires all classes conforming to the interface to provide an implementation for the new method. That is, existing classes that implement Enumeration must implement lastElement because a client may depend on the modified interface. Therefore, this change to the interface does not preserve binary compatibility.

Note that the new method can be expressed by the functionality already provided by the public interface, with an implementation like this:

```
Object lastElement() {
    Object o = null;
    while (hasMoreElements())
        o = nextElement();
    return o;
}
```

While it is easy to provide a default implementation of lastElement for all classes that need it, this does not help. The basic problem is that interface changes must be reflected *immediately* by all classes that implement a specific interface. But these classes are generally not known to the interface developer since they were developed independently by others, and thus they cannot be changed or recompiled.

Interfaces are widely used and generally considered good programming practice, and thus the interface evolution problem is severe. Since any change abandons the support for already compiled code, interfaces are essentially unevolvable. In other words, the interface designer gets "only one chance to do it right".

In contrast, existing classes are somewhat easier to change. For example, adding a new method or field to an existing class is a binary compatible change in Java (although it may cause problems in other object-oriented languages that combine contravariance and genericity).

4.1.2.2 Evolution of Class/Interface Hierarchies

The requirement for binary compatibility also makes it difficult to change a class or interface hierarchy. For example, during the transition from Sun's JDK 1.1.beta3 to the final JDK 1.1, the class java.security.Key was transformed into an interface. This change did not preserve compatibility with pre-existing binaries that depend on Key even though the functionality defined by the interface was exactly the same as in the class (i.e., no methods were added or changed). Nevertheless, the change made it illegal to inherit from Key since it was no longer a class.[2]

In this particular case, the change was possible since it was a change to a beta release that was understood to be unfinished, i.e., programmers did not expect complete binary compatibility with subsequent versions. Had class Key already been distributed widely, the change would have been impossible.

4.1.2.3 Changed specifications

Changing a semantic specification poses a major problem for component evolution. Consider a minor specification faux-pas in Java's Date class which provides a system-independent abstraction of dates and times. While days are numbered from 1 to 31, months are numbered from 0 to 11; i.e., 0 is January, and so on. In retrospect, this numbering scheme was probably a poor design decision. Unfortunately, correcting this flaw would introduce major incompatibility problems, since applications that were constructed for the old Date class would no longer work correctly. To prevent these problems, the specification will probably never be updated, and programmers will have to deal with the unintuitive definition indefinitely, a solution which is not entirely satisfactory.[3]

4.1.3 Summary

We assume that interface and class hierarchies of components must be expected to be slightly inconsistent in relation to each other, even if the individual components are perfectly self-consistent and well-designed. Today's programming languages and environments do not include much support for solving integration and evolution problems, as we have shown in the examples above.

Ideally, a tool for component adaptation should satisfy the following requirements:

1. *No source code access*. Components that come from independent producers are usually only available in object form since software vendors are concerned with protecting their intellectual property. Binaries may also be more compact.

[2] In Java, a class can only inherit from another class; classes cannot inherit from interfaces because interfaces do not contain an implementation.

[3] As a slight improvement, one could add a new function that returns the "correct" month number. But the most intuitive name for this function is already taken, and the presence of multiple functions for the same functionality could further confuse programmers.

2. *Automatic propagation of changes.* A simple operation may require updating many other components to bring the system into a consistent state. Keeping track of all relevant components may be difficult and error-prone and thus should be handled automatically.

3. *Correctness of changes.* Modifications on components may be only valid under certain preconditions, and the tool should ensure that all preconditions are met. For example, a method can only be added as long as there is no name clash with an already existing method.

4. *Ensuring binary compatibility.* Similarly, not all changes preserve binary compatibility. By restricting modifications accordingly, the tool should guarantee that these changes can be reapplied to any future version of the library.

The next section describes how binary component adaptation satisfies all of these requirements.

4.2 Binary Component Adaptation (BCA)

Our approach to component adaptation is based on the insight that binaries distributed in a machine-independent bytecode format often contain enough semantic information to allow safe and effective modifications that operate directly on the binary itself, without needing source code access. In this section we will discuss binary component adaptation and its implementation in detail. We will discuss a Java implementation only, but the concepts should transfer to other languages in a straightforward way.

4.2.1 Overview

The general structure of a binary component adaptation system is quite simple. Component adaptation takes place *after* the component has been delivered to the programmer, and the internal structure of a component is directly modified *in place* in order to make changes. Rather than creating new classes such as wrapper classes, the definition of the original class is modified.

The adaptation system uses two inputs, the original program and a list of changes (adaptations), and produces a modified program (Figure 1). Both the original and adapted program appear in the standard format (i.e., the JVM class file format [LY96]) since they were either produced by a third-party compiler and must be valid inputs to a standard virtual machine implementation. In contrast, the list of changes (which we will call *delta file* since it contains a list of differences or *deltas*) can have any desired format since it is used only by the adaptation system.

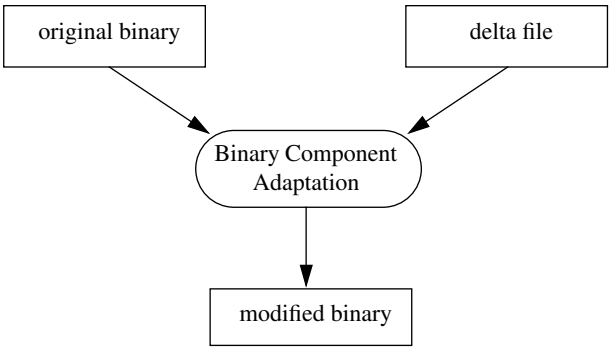

Figure 1. Overview of Binary Component Adaptation

After reading the delta file, the system reads each input object file and produces a corresponding output file that includes all necessary modifications. The virtual machine implementation or native loader later reads these modified files and executes the modified program. By logically interposing the adaptation system between the executable producer (e.g., a standard compiler used by the third-party library provider) and the executable consumer (a standard virtual machine), binary component adaptation requires no changes to either of these systems except in the case of dynamic loading as described further below.

We will use a simple example to illustrate the process. Suppose a programmer needs to rename method foo in class A to bar. The adaptation system must perform two changes to implement the adaptation. First, when reading in class A it updates its method table by replacing the name foo with bar. In Java, this change can be accomplished by modifying the method symbol entry for foo in the constant pool of class A. Second, the system must update all references to foo so that they now refer to bar. In Java, this change also can be accomplished by updating the constant pool of each class that contains references to foo, either because it includes calls to foo or because it is a subclass of A that overrides foo.

In order to preserve consistency, the adaptation system must have access to the complete program, i.e., all class files that could possibly be affected by the modifications. If the complete set of classes can be determined statically,[4] *offline (static) binary component adaptation* simply reads the transitive closure of all classes, producing a new set of class files that forms the complete modified application. Offline adaptation has the disadvantage that it physically duplicates class files and thus increases disk space usage since each application potentially needs its own copy of every class in every library. In return, offline adaptation completely eliminates any runtime overhead since no modifications are needed at load time or during program execution. Furthermore, offline adaptation also allows us to deliver a complete application to a customer without requiring any changes in the customer's setup (i.e., using the customer's existing, standard virtual machine).

Alternatively, *online (dynamic) binary component adaptation* performs all adaptations dynamically, i.e., as classes are loaded during execution. Online adaptation may impose a certain runtime overhead during the loading phase, but code will execute at full speed afterwards. Also, online BCA requires delta files to be distributed with the application, and users must use a BCA-aware runtime system. However, these disadvantage can be compensated by a number of advantages. First, online adaptation requires no additional disk space beyond the space required for the delta file. Second, it does not require a priori knowledge of the set of classes loaded and thus can handle the full Java language including constructs for explicit dynamic class loading. For example, Java's Class.forName method allows a class be looked up by name (causing the class to be loaded dynamically if it is not already present). Since the argument to the call can be an arbitrary string expression, it is not possible in general to determine the transitive closure of classes for a program containing such calls, so that static component adaptation would not be sufficient.

4.2.2 Background: JVM class files

The main requirement for BCA is that a component binary contains enough high-level information about the underlying program to allow inspection and modification of its structure. Java source code is compiled to a such a binary format [LY96] which is portable across different hardware and operating systems. A JVM class file contains bytecodes and a symbol table, as well as other ancillary information required to support key features such as safe execution (verification of untrusted binaries), dynamic loading, linking, and initialization of classes. Unlike other object file formats, type information is present for the complete set of object types that are included in a class file. These properties include the name and signature of methods, name and type of fields (class or instance variables), and their corresponding access rights. All references to classes, interfaces, methods, and fields are symbolic and are resolved at load time or during execution of the program. Since a class references its superclass, the entire class hierarchy can be reconstructed if needed.

Most of this symbolic information is present because it is required for the safe execution of programs. For example, the JVM requires type information on all methods so that it can verify that all potential callers of a method do indeed pass arguments of the correct type. Similarly, method names are required to enable dynamic linking. Indeed, it is likely that a file format that supports type-safe dynamic linking already contains enough information to support at least some form of binary component adaptation. The exact extent to which binary component adaptation is possible depends on the amount of symbolic information that is available; each item potentially changed by binary component adaptation must be described symbolically. In the case of Java, the JVM file format contains enough information to allow a wide range of modifications.

[4] This assumption is often referred to as the "closed world" assumption.

4.2.3 Delta Files

The delta file describing the required adaptations should be compact and efficient to read, especially in the case of online adaptation. Since delta files are generated by tools rather than edited by hand, they can be highly encoded. In general, delta files will contain embedded code fragments, e.g., when adding a new method to a class.

The component programmer could generate delta files by specifying the adaptations by using a specialized tool (i.e., graphical editor) or by creating a textual adaptation specification that is subsequently compiled into a delta file by a compiler-like tool. For example, an adaptation specification for the String example in section 4.1.1 could look like this:

```
delta class java.lang.String {
    add method
        public String encrypt() { ... }
        public String decrypt() { ... }
}
```

A natural extension to delta file generators are tools to manipulate deltas, e.g., to decompile them into a textual or graphical representation, or to allow delta files to be modified or merged. The latter functionality may be particularly useful since it allows deltas to be reused and combined. For example, a delta that adds printing to a class may be combined with a delta for serialization, resulting in a combined set of modifications.

As a language construct, deltas are similar to *mixins* as defined by Bracha [Bra92], although mixins are part of the programming language, work at the source level, and do not modify classes in place. Many of the operations defined on mixins (especially composition and inheritance) could also be defined on deltas, turning adaptation specifications into first-level constructs of a full-fledged adaptation language.

4.2.4 Potential Adaptations in Java

The range of possible adaptations is limited only by two constraints: the amount of symbolic information available in binaries, and the desire to enforce binary compatibility. In the case of Java, class files contain enough symbolic information to allow virtually any change, so that the first constraint does not apply.

Table 1 lists the wide range of useful changes to a Java class or interface that are possible. In addition, a number of other modifications could be implemented whose usefulness is currently less clear. For example, visibility attributes (private, public, etc.) can be changed, although some changes may not preserve binary compatibility. One could also add new parameters to an existing method, providing default values to be passed from existing call sites.

Possible Adaptation	Example Use
add method or instance variable to class	integration or customization (see example in section 4.1.1)
extend list of interfaces supported by class	integration or customization
add superinterface to interface	type hierarchy restructuring for better integration
rename class, method, or instance variable	name conflict removal
move method to superclass	restructuring
split class into two classes	restructuring
add method to interface	component evolution (see example in section 4.1.2.1)
convert class to interface	component evolution (see example in section 4.1.2.2)

Table 1. Possible adaptations in Java

4.2.5 Enforcing Binary Compatibility

Component adaptations must preserve a component's type correctness in order not to invalidate existing subtype relationships. More precisely, the adaptation must result in a program that still satisfies all semantic rules defined by the underlying language (e.g., Java).

These constraints can be expressed by a set of preconditions for each modification. For example, a class can only be renamed if no name clash occurs. The constraints are relatively straightforward to derive from the language definition.

In addition to these semantic constraints, component adaptations should *preserve binary compatibility*, unless the programmer is willing to forgo its advantages. Informally, binary compatibility implies that a working application using component C will continue to work with the next release C' of the component (a more detailed description of binary compatibility can be found in [F+95]). To extend compatibility to adaptations (deltas), a delta for component C must be compatible with a new component release C' if C' is compatible with ordinary clients of C. In other words, if a component release preserves binary compatibility with ordinary clients, we want to guarantee that adapted components also remain compatible. In Java, a new release of a class is compatible with existing clients as long as its public interface C'_{pub} includes C_{pub}. That is, adding methods and fields to a components is fine, but changing or removing members can lead to incompatibilities.

Informally, the binary compatibility rules require deltas to depend only on the public items (classes, variables, methods, etc.) of a component, so that the delta can be applied to any compatible future release of the component. Therefore, the compatibility rules ensure that adapted binaries are at least as compatible with future releases as ordinary clients would be.

More precisely, assume a programmer has created an adaptation δ for a component C that consists of private and public parts C_{priv} and C_{pub}, and that δ modifies only C_{pub}. Thus, $\delta(C_{priv} + C_{pub}) = \delta(C_{priv}) + \delta(C_{pub}) = \delta(C_{pub})$ since $\delta(C_{priv})$ is empty. If the component producer now delivers a new release C' of C, $\delta(C') = \delta(C'_{pub})$.

If the new release does not change the component's public interface, the adaptation will still be valid for C' since $C'_{pub} = C_{pub}$ and the adaptation was valid for C_{pub}. In other words, prohibiting modifications to C_{priv} guarantees compatibility of deltas with new releases that do not change the public interface of C.

Note that even though the programmer is not allowed to make changes to private methods, the BCA implementation may modify their code. For example, if the programmer renamed the public method foo to bar, BCA rewrites all calls to foo into calls to bar, regardless of whether these calls are from public or private methods. However, this rewriting cannot possibly create semantic conflicts.

But even if a delta operates only on the public members of a class, the modifications may still create indirect dependencies on non-public members since these are in the same scope. For example, an adaptation that adds a public method print could introduce a conflict when applied to a future release that adds a new non-public method print with the same signature. (If the two methods had different signatures, no conflict would exist since Java allows overloading.) Fortunately, this conflict is simple to detect when the class is loaded. Renaming can resolve the conflict: as long as the BCA implementation can identify, for each call site, which of the two methods it calls, it can simply rename one of the methods (and rewrite the affected calls) to resolve the conflict without requiring programmer intervention.

Online adaptation poses an additional problem since the class loader may encounter both classes that call the print method introduced by the delta as well as classes that call the new print method provided in the new release. (The latter could be classes that were updated themselves as part of the new release.) How can the system tell whether a particular call site was compiled against the adaptation created by the component user or was compiled against the new component version (and thus refers to the new method introduced by the component vendor)? To solve this problem, we annotate each client of an adapted binary in order to identify all accesses of the client to adapted features. We currently express this declaration with a class constant in the client class whose contents are inspected during class loading.[5] Thus, the system can recognize calls in

such classes as referring to the method introduced by the delta. No class can unknowingly contain calls to both print methods, since such a class would have been compiled against both the adaptation and the new component release, and this compilation would have detected the conflict.

The above discussion showed that deltas preserve binary compatibility if $C'_{pub} = C_{pub}$, i.e., if the public interface remains unchanged. This level of compatibility is already useful, but we would like to extend the compatibility guarantee for deltas further, to the point where delta compatibility is implied by the compatibility of the base component. In other words, deltas must preserve compatibility even if a future release adds methods or fields to the component. Such a new release can create a name conflict if it adds a new public method with the same name and signature as a method added by a delta. Fortunately, renaming can again resolve the conflict since it is just a variant of the potential conflicts with non-public methods described above. If a programmer needs to write a class that uses both the adaptation and the new (conflicting) features of the new release, he or she must first resolve the conflict by renaming the method in the adaptation, i.e., by creating a new delta.

4.2.6 Summary

Binary component adaptation fits well into systems that use binaries that retain (some) high-level information, such as Java's class files. In particular, Java's class files contain enough information to allow the complete Java structure to be modified: fields, methods, classes, and interfaces. Adaptations can be performed offline (if the transitive closure of all classes is known and available) or online during class loading. To ensure binary compatibility, modifications must depend only on the public attributes (classes, variables, methods, etc.) of a component, so that the adaptation can be applied to any compatible future release of the component.

4.3 Discussion

4.3.1 Examples revisited

Let us first explore how the example problems of section 4.1 can be solved with binary component adaptation. To integrate components A and B from section 4.1.1, both classes require a common supertype Printable; in addition, class A needs a print method. These changes result in the following delta specification:

```
delta class A {
    add interface Printable;
    add method
        public void print() { output(System.out); }
}
delta class B {
    add interface Printable;
}
```

Both changes retain binary compatibility: adding a new method cannot affect existing components since they do not refer to this method, and adding an interface is similarly transparent to code that does not use this interface.

The interface evolution problem in section 4.1.2 is easy to solve as well. To add the lastElement method to the Enumeration interface, we just need to provide a default implementation in the delta, to be added to any class supporting the old interface:

[5] Sun has taken a similar approach to include versioning information in some JDK classes.

```
delta interface Enumeration
    add method
        public Object lastElement() {
            Object o = null;
            while (hasMoreElements())
                o = nextElement();
            return o;
        }
}
```

This adaptation specification extends interface Enumeration with the additional method declaration and implicitly performs an add method adaptation to every class that supports Enumeration (or any subinterface of Enumeration).

The problem with the changed definition of the month method in the Date class can also be solved with binary component adaptation, although the solution is somewhat more complicated. The basic idea is to rename the old definition to obsolete_month and then to add the new definition of month:

```
delta class Date {
    rename method month to obsolete_month;
    add method
        public int month() {
            return obsolete_month() + 1;
        }
}
```

A complication arises because the old and new month methods have identical signatures. Therefore, it is unclear whether a particular call site should be redirected to obsolete_month or whether it was compiled with the new definition. However, the simple annotation of the class files with versioning information (described in section 4.2.5) solves this problem.

4.3.2 Deltas as First-Level Components

Since deltas are central to component reuse in a BCA-oriented model, the manipulation and management of deltas becomes an important issue. Space restrictions prevent a full discussion in this proposal, and thus we only summarize this point.

The basic question is, does BCA allow a delta to be applied to an other delta? Yes, because non-conflicting deltas can be composed, i.e., $(\delta_1 \otimes \delta_2)(C) = \delta_2(\delta_1(C))$ just as with function composition. (The order is significant: the changes of δ_1 are applied first, then the changes of δ_2.) Composing deltas is quite useful because it allows deltas to be reused and combined. For example, a delta that adds printing may be combined with a delta that offers serialization. Naturally, conflicting deltas cannot be combined without manually resolving the problem.

4.3.3 Distribution of Deltas

With offline BCA, delta files reside only in the developer's workspace and do not need to be distributed to users since the modifications specified in the delta files are already reflected in the set of object files produced by offline BCA. For online BCA, however, delta files become part of the distributed application because adaptation happens at dynamic load time, i.e., while the application is running.

The simple BCA implementation sketched in Figure 1 modifies class files before they are passed on to the native loader. This organization is very simple to implement since the presence of modifications is invisible to the standard virtual machine (VM), and thus the VM needs only minor modifications in order to insert BCA into the loading process.

However, this organization requires that all deltas be known at start-up time. An example will illustrate why: suppose we have renamed method foo in class A. Now, even before A is loaded, another class B may be loaded that contains (unresolved) references to A. These unresolved references are possible because in many systems (including Java), classes are loaded only when actually needed, in order to minimize the set of

classes that is loaded. If BCA wasn't aware of the renaming of foo when loading class B, the references to A would not be rewritten correctly.

This example allows us to define the load time restriction for deltas more precisely: the delta for A does not need to be loaded until the first client of A is loaded, i.e., the loading of the delta file for A can be deferred until the first reference to A is encountered. Therefore, deltas do not need to be combined into a single delta file for the application but could remain as per-class deltas. For example, Java class A would have an A.delta file in addition to the A.class file. Although this setup is relatively dynamic, delta files will still be loaded more eagerly than the class files.

This disadvantage could be eliminated by integrating BCA into the virtual machine (VM). In this scenario, a delta file for a class would not be loaded until the class itself is loaded, requiring the VM to "patch up" unresolved (and as yet unmodified) references to foo once class A has been loaded and its delta is known. Such a system is safe as long as unresolved references can always be modified, since foo could not have been previously called because this use of A would have required A to be loaded earlier.

In either case, delta files could be distributed just as the class files themselves are distributed today. This includes distribution over the Internet, where the delta files could be loaded from the same server as the class files, or from a separate adaptation server if desired.

Given some versioning support in the base system, applications could also include conditional deltas to be used if the local environment only has an earlier release of a particular library. To enable this functionality, BCA must be able to determine what version of a library an application needs and which version is installed. Similar to dynamic component adaptation [MS97], BCA could dynamically compute which deltas are required to bridge a particular version mismatch.

4.3.4 Limitations

As already discussed, BCA works only if object files contain enough symbolic information. In addition to descriptions of classes, methods, and instance variables, the object files must also contain information that allows BCA to find all references to classes, methods, etc. so that they can be changed if necessary. To support the addition of instance variables, BCA may also need to identify allocation sites in order to update object sizes.

BCA is also intimately tied to versioning, requiring version information for some adaptations as discussed in section 4.2.5. Also, while deltas on deltas are possible as described in section 4.3.2, if they are used too extensively the integration and evolution of deltas themselves may become a new problem. However, as long as most deltas are not widely shared (since they perform very specific adaptations on behalf of a single application) these problems should be easier to handle than the integration and evolution of widely distributed components. Also, the ability to perform late adaptations remains a valuable help in any case.

4.4 Related Work

Much previous work has viewed the problems of component adaptation and evolution as a programming language problem, and several language features have been proposed that can address some (but not all) problems. *Descriptive classes* [San86] allowed the programmer to create supertypes after the fact, solving part of the example problem in section 4.1.1. The usefulness of creating new superclasses for existing classes was discussed in detail by Pedersen [Ped89] and implemented in Cecil [Cha93a]. *Predicate classes* [Cha93b] offer yet another way to extend and adapt objects, although they were conceived for a different purpose. In languages allowing multiple dispatch (e.g., CLOS and Cecil) new methods can be attached to existing classes.

Horn's *enhancive types* [Hor87] do not alter types (classes) directly but instead allow a base type to be coerced into another type (the enhanced type) that offers additional methods implemented in terms of the base type's public interface. Unlike BCA, enhancive types do not allow the addition of instance variables or the renaming of methods or classes. Remarkably, enhancive types respect compatibility, i.e., no existing code

needs re-typechecking because of the enhancements, and thus they can be applied to future versions of the code without problems. Since existing code is left untouched, however, the range of permissible modifications is much smaller than with BCA.

Objective-C's *categories* are named collections of method definitions that are added to an existing class [ObjC96]. Rather than defining a subclass to extend an existing class, category methods are added directly to the class. As with subclassing, there is no need for source code for the extended class. Thus, categories allow class adaptations similar to binary component adaptation. However, changes are limited to adding and overriding methods; a category can't declare any new instance or class variables or rename a method. When a category method overrides an existing method, the new version cannot invoke the method it replaces. Finally, the Objective-C compiler cannot guarantee compatibility with future versions of the class.

In previous work [Höl93], Hölzle discussed the shortcomings of language-based solutions to component integration and observed that many problems could be solved if types could be modified in place. He also proposed to deliver executables in a higher-level format to allow such modifications, but since no language supported a standardized high-level executable file format at the time the idea seemed impractical.

Palsberg and Schwartzbach recognized similar problems with traditional reuse mechanisms and proposed a type substitution mechanism aimed at maximizing code reuse [PS90]. However, for adaptation purposes, their solution is both too general and too restricted: too general because in the presence of subtype polymorphism it may require the re-typechecking of a component's implementation, and too restricted because it does not allow adding new methods or renaming methods.

Nimble [PA91] is a tool for procedural languages that allows programmers to transform actual procedure parameters at run-time. A map defines the rules of parameter conversion and is used to generate an adaptor which is linked into the application, performing the parameter translation at run-time. Although Nimble does not require source code and can bridge simple interface mismatches, it is much more restricted than BCA since it only addresses parameter type conversions.

Bracha [Bra92] presents a framework for modularity in programming languages, viewing inheritance as a mechanism for module manipulation. Bracha proposes *mixins*, a more powerful form of inheritance since a mixin is not inextricably bound to a parent. A mixin is a function from classes to classes, parametrized by a parent which it is modifying. In our context, a delta can be regarded as a mixin which is then applied to a class. Unlike mixins, deltas modify the class in place rather than producing a new class. Even though Java does not support mixins at the language level, they could be simulated using binary component adaptation.

An alternative adaptation strategy to BCA, *dynamic interface adaptation* [MS97] dynamically loads adaptors (wrappers) to bridge interface mismatches between Smalltalk components. Adaptors (or wrappers) implement a mapping function between call-out and call-in interfaces of components. If the interfaces between components show mismatches, the run-time system looks up an appropriate adaptor and configures it on demand. Yellin and Storm [YS97] define adaptors that in addition to interface mismatches can also bridge sequencing constraints (protocols). Unlike BCA, adaptors do not modify classes in place and usually slow down calls to adapted components.

BCA could be used to support program refactoring. For example, Opdyke [Opd92] defines a set of restructuring operations (refactorings) to support the design, evolution and reuse of object-oriented application frameworks. Refactorings do not by themselves change the behavior of a program, but they restructure it in a way that makes the software easier to extend and reuse. Similarly, Hürsch [Hür95] presents a framework for evolution that automatically maintains the overall consistency of an object-oriented system. Both systems could operate on binaries using BCA instead of transforming source code.

Forman et al [F+95] discuss release-to-release binary compatibility between class libraries and its importance for software distribution. IBM's System Object Model (SOM) guarantees that new methods and classes can be added to a SOM library without recompiling client applications. SOM also supports evolution of class libraries through a large number of compatibility preserving transformations, but all the transforma-

tions require source code availability. However, it would be conceivable to construct a BCA system for SOM binaries.

Adaptable binaries [G+95] also allow direct transformations on a binary, but unlike BCA the adaptation is performed at the instruction level, i.e., it ignores the programming language semantics. Its main applications are quite different from BCA and include software-fault isolation (software memory protection), machine-retargeting, and optimizations.

A number of higher-level binary distribution formats (e.g., ANDF [OSF91], Omniware [A+96], Slim Binaries [FK96], and BRISC [E+97]) could be extended to support BCA. In addition to a description of all classes and types, the object files need enough information to allow dependent code to be updated (e.g., because dispatch tables or object sizes changed).

Harrison and Ossher [HO93] as well as Smith and Ungar [SU96] argue for subject-oriented programming, i.e., an object model that allows multiple different perspectives of the same object. Ideally, different (subjective) views of the same object can coexist simultaneously in a single application. Binary component adaptation does not go quite that far, allowing only one application-specific version of a class. Subject programming also assumes full source-level access at integration time and does not necessarily preserve binary compatibility, and views subjectivity more as a programming language level mechanism than as a binary rewriting tool.

4.5 Plan of Work

A prototype of offline BCA for Java is already working, implementing most of the adaptations described in section 4.2.4, and we have made enough progress towards an online BCA prototype to be convinced that an efficient implementation is possible. We propose to implement online BCA for Sun's JDK VM because this VM is popular, freely available, and available in source form.

The proposed implementation consists of several components:

- A graphical user interface that allows programmers to create deltas by editing existing class interfaces.
- A semantic checker that verifies the legality of the programmer's modifications and checks them for binary compatibility.
- A compiler that produces compact delta files from these modifications.
- A modified class loader that adapts classes as they are loaded, based on the modifications described by the delta files.
- A set of auxiliary higher-level tools, such as tools to merge, compare, and manipulate deltas.

We are planning to work on the core technical issues first, in order to detect any design problems as early as possible. Table 2 shows an overview of the proposed work schedule. During the first year, we plan to design and implement the class loader and semantic checker, plus a first version of a delta compiler for generating delta files as test input. At the end of the year or the beginning of the second year, we plan to release this initial system to the public in order to obtain feedback on its usefulness. During the second year, we will also perform experiments on larger software systems selected from other Java projects at UCSB in order to evaluate the range of applicability of BCA. (Several large Java applications are currently being constructed by other research groups in the department.) During the second year, we will also implement the graphical user interface and the auxiliary tools. In addition, we will also tune the performance of the class loader in order to minimize the run-time overhead of component adaptation. We plan to make the implementation publicly available in order to facilitate experimentation with binary component adaptation.

As outlined in Table 2, we think that the proposed research can result in the bulk of a Ph.D. thesis and two M.S. theses. We have budgeted two quarters and one summer for each of the M.S. students (M.S. thesis typically require six months of full-time work), and support for the Ph.D. student throughout the year.