Extending the Notion of Type Conformance to Interfaces and Type Systems

Position Paper for OOPSLA '93 Workshop on

Supporting the Evolution of Class Definitions

Barbara Staudt Lerner
Computer Science Department
University of Massachusetts
Amherst, MA 01003
blerner@cs.umass.edu

July 1993

1 Motivation

Class evolution is a normal aspect of the maintenance of object-oriented programs. While class evolution is closely related to the schema evolution problems faced by traditional database systems, there are two fundamental differences important for this discussion. First, the fact that class definitions are frequently shared directly in multiple applications rather than shared via "cut-and-paste" as in relational database schemas implies that modifying a class may impact many programs, and the class maintainer may not even be aware of all affected programs. This difference motivates the need for upward compatibility so that programs using a modified class can continue to do so unaffected by the changes.

Second, the fact that classes are more loosely connected than the relations in a relational database schema offers hope that we can address evolution in a more modular manner than with traditional databases. Many evolution systems take this to an extreme and treat evolution as a problem only involving individual classes, However, a maintenance activity might simultaneously affect multiple classes and require a more unified solution than is possible when considering each class's evolution in isolation.

In this paper, we discuss our approach to upward compatibility and changes involving multiple classes. Before doing so, we identify the basic tenets upon which our ideas of class evolution are based. These tenets are not meant as absolutes, but they certainly bias the research directions that we have taken. The basic tenets are:

- Newer is better.
- Newer versions generally add functionality and rarely delete any.

The first tenet states our belief that newer versions of classes are generally superior to older versions. This may include improved functionality, reliability, performance, etc. As a result, when class instances are

shared by programs developed with multiple versions of a class definition, we are biased towards converting class instances to the version used by newer versions of classes.

The second tenet states our belief that evolution much more frequently involves extending the functionality of a class rather than reducing its functionality. Newer versions of classes often include all the functionality provided by earlier versions. As a result, converting objects to a newer version rarely reduces their usefulness for old code.

The general approach to class evolution that we propose involves converting objects to a newer version when new code accesses an old object and screening new objects when accessed by old code.

2 Class Evolution

In this paper, we assume a language that defines classes as consisting of an interface and an implementation. The interface contains only the signatures of the public methods, both those defined locally and those inherited from superclasses. The implementation contains the declarations of the class's instance variables, the private methods, the bodies of the public methods, and the list of superclasses.

Classes can be organized into two hierarchies: a type hierarchy and a class hierarchy. A type hierarchy organizes classes according to their interfaces. Class A is a subtype of class B if for every method in B's interface there is a method in A's interface that is a subtype of the method from B using the contravariant rule of subtyping. The type hierarchy is used to support type checking. It is not necessary to declare the type hierarchy explicitly since it can be inferred from the class definitions.

A class hierarchy organizes classes according to their implementations and is used to support inheritance. The class hierarchy is explicitly declared by the programmer by identifying the superclasses for each class. A subclass inherits the variables and methods defined and inherited by its superclasses. Conflicting names must be resolved, although the exact method for resolution is irrelevant for this paper. Since the class hierarchy is used only for inheritance and not type checking, the programmer may override any definition with any other definition. In particular, there are no typing constraints on overriding, although failure to follow the contravariant rule when overriding public methods will result in a subclass that is not a subtype of one or more of its superclasses.

We classify class changes into the following three categories:

- Interface changes
- Implementation changes
- Class identity changes

Interface changes are those changes that involve modifications to the public portion of a class. Changes may involve adding a method, changing the name of a method, modifying a parameter list, or occasionally, deleting a method. These changes may occur either locally or as a result of modifying the class hierarchy.

Implementation changes are changes to the private part of a class. They include changing the instance variables, modifying the body of a method, or modifying the signature of a private method. These changes may occur either locally or as a result of modifying the class hierarchy.

Class identity changes involve creating, deleting, or renaming an individual class, or reorganizing classes in such a way that there is no single unique new class corresponding to each old class.

3 Type Conformance

Type conformance determines how a class may change from one version to the next and still be managed by a system's support for class evolution. One solution to type conformance requires that new versions of classes be upwardly compatible with the versions they replace. In an object-oriented setting, this translates into creating new versions that are subtypes of the versions they replace. The reasoning behind this is simple: any objects created with the new version of the class can be manipulated by programs expecting the old version. New programs, which can be aware of the old versions, can use objects created with those old versions by inserting a screening operation that makes the object appear as if it was created with the new version of the class. Many systems do not strictly follow this subtyping rule, however, and allow variables to be deleted during evolution. Since deletion violates the subtyping rule, changes involving deletion are not upwardly compatible. Therefore, changes involving deletion also require a screening operation to allow old code to work with new objects or else require all code using a class to be updated when the class's definition changes.

Another limitation of existing systems is that the type conformance they support typically requires that each old class have a single corresponding new class. Type conformance therefore restricts a programmer to making changes local to an individual class. Most systems further restrict the kinds of changes that can be made. Also, type conformance typically relies heavily upon the class's implementation, not just its interface.

For example, here are the limitations imposed by some systems on changing the type of an instance variable. In Orion [BKKK87] it is possible to change the type of an instance variable to be a supertype of its old type, but not a subtype or unrelated type. In GemStone [PS87], it is possible to change the type of an instance variable arbitrarily. However, if an instance variable is subtyped, any values that are not in the subtype are deleted. Similarly, if an instance variable is changed to an arbitrary type, no attempt is made to preserve the old value or transform it into an acceptable value. O_2 [Bar91] goes further than either of the previous systems by allowing the programmer to define a transformation function to call when the type of an instance variable is changed to something other than a supertype. However, O_2 still deals only with changes local to an individual class and bases its conformance rules on a class's implementation.

In this paper we discuss two ways of generalizing the notion of type conformance to provide more flexible evolution. First, we discuss type conformance based on class interfaces rather than implementations. Next we discuss a more general notion of type conformance that allows a single class to be replaced with multiple classes, or several classes to be combined into one.

3.1 Interface-Based Type Conformance

If classes interact through well-defined interfaces, then it should be straightforward to support evolution when a class's implementation changes, but its interface remains the same. This requires that we be able to select a method to execute not just based upon the class of the object to which it was sent, but also based upon the version of the class used by the object. We could accomplish this either by relinking all programs using a modified class to include the code for all versions, or by storing the object code associated with the methods of all versions of a class in the object base to be dynamically linked in when necessary.

Furthermore, class changes that result in upwardly compatible interfaces can also be supported transparently. Upward compatibility for a class interface requires that new versions of a class be subtypes of old versions. Since the only things visible in the interface are the signatures of the public methods, we require that each method in the old version have a corresponding method in the new version that is a subtype using the contravariant subtyping rule. In this way, old code is able to interface with new objects through the

modified interface. even though the variables and method implementations may have changed arbitrarily. Of course, if the new implementation modifies the semantics of the methods, the behavior may not be what the old code is expecting. The only thing that is guaranteed is that the new version is type-safe to use with old code.

The changes to a class interface that are upwardly compatible are:

- Adding a method
- Renaming a parameter
- Changing the type of a parameter to a supertype
- Changing the type of the return value of a method to a subtype

While the ability to arbitrarily change the class's implementation gives the maintainer a wide latitude in the kinds of changes that can be made, it is still possible that a maintainer may want to change an interface so that it is not upwardly-compatible. Such changes involve one of the following:

- Deleting a method
- Renaming a method
- Deleting a parameter
- Adding a parameter
- Changing the type of a parameter to a subtype or unrelated type
- Reordering parameters
- Changing the type of the return value of a method to a supertype or unrelated type.

To support non-upwardly compatible interface changes, we introduce the concept of an *obsolete interface* and a *current interface* to the class interface. An obsolete interface adds the necessary methods to make the new interface upwardly compatible with the previous version. The obsolete interface is only intended for old code to use. The current interface defines the interface that new code should use.

For example, even though a new version might delete a method, following our belief that functionality is rarely deleted, the same functionality is probably still available but is accessed in a different way. If this is true, it would be possible for the maintainer to include the "deleted" method in the obsolete interface and then implement it in terms of methods in the current interface. In the rare cases where a method is truly deleted, it would be flagged as such in the obsolete interface and attempts to execute it would result in a runtime error. If the maintainer wants to continue to support an obsolete method, but requires the use of private methods to do so, he/she should reconsider whether that functionality is really obsolete.

As a class evolves it collects a series of obsolete interfaces. Once an obsolete interface (and its implementation) is created, it is never necessary to modify it (except to fix errors). Each obsolete interface simply provides a screening for objects of the next version to appear as objects of an older version. An object that is several versions newer than the calling code requires should be passed through the sequence of screening functions required by each intervening version.

The compiler can ensure that new code only uses the current interface to a class and not obsolete interfaces. A more interesting situation arises when an existing class that uses a modified class is updated.

We could imagine a situation in which the modified class is "grandfathered" and allowed to continue to use the obsolete interface. However, the programmer should certainly be informed that he/she is using an obsolete interface, and in particular should be told of methods that are not supported in the obsolete interface. A better solution would be an evolution system that supported the update of the modified code to the current interface. If the implementation of an obsolete method uses only public methods, all calls to that obsolete method could be replaced with the implementations of those obsolete methods in-line. The evolution support system should be able to automate the in-lining, although the resulting code might not be optimal. If the obsolete method uses private methods in its implementation, the maintainer of the client class is burdened with updating its implementation to correspond to the current interface. Again, we feel this is appropriate because class evolution should only be applied when a maintainer truly intends to replace a class definition with a newer one, and should not be used to create a collection of variants of a class.

This approach of maintaining multiple interfaces to a class is similar to the approach proposed by Skarra & Zdonik [SZ87, Zdo90] and Clamen[Cla92]. A major difference is that we treat the current version of a class as special and tend to migrate objects toward the current version. In contrast, Zdonik and Clamen treat all versions as equally valuable and maintain multiple versions of objects as well as their classes in order to support evolution. Also, we hide implementation changes behind a class interface, while Skarra & Zdonik and Clamen provide multiple versions of objects when an implementation changes as well. Obsolete interfaces are also related to obsolete methods provided by Eiffel [Mey90]. However, in Eiffel, an obsolete method can always be used, even by new code, although the compiler warns when one is. Eiffel offers no other versioning support nor any support in replacing uses of obsolete methods.

3.2 Multi-Type Conformance

Obsolete interfaces address the issue of how code written using an old version of a class definition can operate on objects created with a newer version. We also need to solve the problem of newer code working on older versions of objects. If the interface is not changed, there is no problem; code written to use a newer interface is still able to access old objects using the old interface and implementation. However, if the interface is changed it most likely provides functionality that is not available in the old interface. Code written with the new interface may try to call this code, which is not provided by the versions used by old objects. Since we believe that the newer version was created as an improvement over the older version, we transform the old object to the new version in this case.

We can look at systems such as Orion, GemStone, and O_2 for inspiration on how to support conversions of old objects to new. However, we find their mechanisms lacking in that they support evolution only in terms of its effects on individual classes, rather than thinking about evolution in terms of an entire type system. Also, they focus on the type safety properties of the object base and ignore issues of data integrity that can occur when changes incorporate new semantics. In this section, we discuss evolution from the point of view of the implementations since we are interested in addressing the issues of converting old objects to newer representations as defined by their instance variables. The obsolete interface mechanism can continue to be used to allow old code to access new versions even when a change affects multiple types.

There are several common type system changes that can affect multiple types. To support evolution involving these categories of changes, we may need to be able to convert or screen multiple objects in our object base to make them appear as a single new object of the appropriate version or to allow one old object to appear as an instance of more than one new class. The problem of converting instances to new versions is significantly more complicated when multiple types are affected than when one considers only changes local to individual types. The intent of this section is to point out the necessity for multi-type conformance,

rather than offer a complete solution.

The multi-type changes discussed here provide examples of reasonable changes that a maintainer might want to perform, yet are beyond the capabilities of current evolution support. The multi-type changes that we have identified thus far consist of the following:

- *Encapsulating* part of the functionality of one class to create a new class.
- *Specializing* a class by creating one or more new subclasses.
- *Moving* functionality/variables from one class to another existing class.
- *Combining* some or all of the functionality in two or more classes into a single new class.

Multi-type changes also raise an issue of multi-object conversion. Three situations requiring multi-object conversion may arise:

- An old object may be replaced with several new objects.
- Several old objects may be replaced with one new object.
- Several old objects may be replaced with several new objects.

Of these, the last two are the greatest cause for concern. In these cases, conversion requires us to identify a collection of old objects in order to create the appropriate new object(s). This requires an ability to describe what those related objects are and to find them when we want to do the conversion. A powerful query language would simplify these types of conversions.

Encapsulation occurs when a new class is created by dividing the variables (and functionality) of an existing class into two separate parts. The result is that an old object becomes split into two objects during conversion, although it is likely that one of the objects will be the value of an instance variable in the other object. For example, suppose we have the following class:

```
class BankAccount is
    acct_type: AccountType;
    customer: Name;
    customer_address: Address;
    customer_phone: Phone;
    amount: Dollars;
    ...
end class;
```

This class may include methods to get the customer's name, address, etc. We might want to reorganize our classes so that we have a single class of customer information with a component for each account rather than duplicating customer information everywhere.

```
class Customer is
    customer_name: Name;
    customer_address: Address;
    customer_phone: Phone;
    accounts: list of BankAccount;
    ...
end class;

class BankAccount is
    acct_type: AccountType;
    the_customer: Customer;
    amount: Dollars;
    ...
end class;
```

To support object conversion, the evolution system must be able to create the necessary customer objects by extracting information from the BankAccount objects. This should be straightforward. A more difficult problem is that we want to avoid having duplicate Customer objects for each individual customer. This is an example of a several-several mapping. When we convert a single BankAccount object, we actually need to convert all bank accounts for that customer at the same time in order to ensure the integrity of our object base. One can imagine other encapsulation examples in which the transformation is 1-several, so that it would be unnecessary to test for duplicates.

An interesting observation about this example is that it demonstrates that adding a new class is not necessarily trivial with regard to the evolution support required. Of course, the addition of the class is not the source of the problem, but rather the use of this new class in a new version of an existing class is. However, as we noted earlier, the most common solutions restrict changing variable types to supertypes or subtypes of the old type, and would prevent a maintainer from implementing this change altogether.

Specializing occurs when an old class is replaced by several new classes. For example, a class might be replaced by several more specialized subclasses. Consider how the BankAccount class could be modified and replaced by more specialized classes:

```
class BankAccount is
    the_customer: Customer;
    amount: Dollars;
    ...
end class;

class SavingsAccount inherits from BankAccount is
    ...
end class;

class CheckingAccount inherits from BankAccount is
    ...
end class;
```

In this case we might want to replace all old instances of BankAccount with either an instance of SavingsAccount or CheckingAccount. In other cases, we might want to allow some instances to remain instances of the superclass. In either case, we want to be able to determine which old bank account instances correspond to which of the new subtypes in order to maximize the usefulness of those account instances by new code intended for the subtypes. This introduces the need for our conversion functions to perform conditional conversion. Note that the new classes would probably not be subtypes of the old class. The old class would have included a method to determine what type of account it was, while this is implicit in the type of the new classes. The new classes therefore need trivial functions in their obsolete interfaces to return their type.

A moving change occurs when some functionality/attributes are moved from one class to another. Most commonly this movement would occur between two classes where one class is used as the type of an instance variable in the other. This is another example of a several-several mapping. To support object conversion to the new version, the evolution system would need to be able to identify which pairs of objects were affected by such a change so that the appropriate variable values could be moved between them. Unlike the previous example, this change is inherently several-several.

A combining change occurs when the new class merges functionality previously found in multiple classes. For example, suppose a university department has an object base that contains information about the members of the department. The Member class might have subclasses consisting of Employee and Student. Since some students might be employed by the department there may be two objects representing an individual. A reorganization of this hierarchy could include creating a StudentEmployee class to hold the information pertaining to those people. In this case, the new class would probably be a subtype of the two existing classes, making the upward compatibility problem trivial. This is an example of a several-1 mapping. In addition to identifying which objects in an existing object base should be paired together to form instances of this new object type as we did earlier, we also face the problem of updating references to the two old objects so that they point to this new object. We want the new object to be reachable by any reference to either old object, effectively giving the new object two object identifiers. Most existing systems would only introduce the new class, but would not transform any existing instances. While this is a type-safe solution, it violates the integrity of the object base because it does not model the real-world objects as accurately as it should.

In related work, Casais [Cas90] describes algorithms to analyze a class hierarchy and reorganize it to make the relationships between classes clearer. These types of reorganization simultaneously affect multiple classes. Since these changes are driven by an algorithm, it should be easier for an evolution system to understand their intent than when similar changes are performed by a maintainer directly. As a result, it should be possible to automate updating the objects appropriately when the class changes themselves are automated, although Casais does not address this issue.

OTGen [LH90] provides support for automatically generating transformation functions by observing the changes that a maintainer makes to a set of type definitions. While it does not automate transformations involving multiple changes, it provides a declarative language that makes it simple for a maintainer to specify many types of transformation functions and thus provides a framework to support multi-type changes. However, even with this intervention from the maintainer, OTGen is still limited in its support for multi-type changes since it lacks a query facility and does not support objects with multiple identities.

4 Conclusions

Most existing support for class evolution is extremely limited. First, most systems focus on the changes to the representations of classes specified by their instance variables, rather than on the interfaces to the

classes. As a result, many changes that could be neatly hidden behind a clean interface become directly visible and are prevented by the type conformance rules of the evolution system. Second, the changes that most systems support are rather narrow in their focus and only treat classes in isolation, rather than dealing with type systems as a whole. The result limits the usefulness of the evolution support by either prohibiting useful changes or supporting them at the type level, but not the instance level.

This paper offers a solution to the first problem by outlining an approach in which a language supports evolution via obsolete interfaces to classes. This approach offers the maintainer a great deal of flexibility, but still requires him/her to face the issues of upward compatibility and to explicitly think about the effects of evolution when modifying a class.

The second problem of supporting multi-type changes requires more support from the object base system. First, support for multi-type changes requires a powerful query facility to allow collections of objects to be located that must be converted simultaneously. Second, it requires the ability for conversion functions to make transformations conditional on the state of an individual object to support specialization. Third, it requires an object to be able to take on two object identifiers to allow old objects to be merged. Also, since most of these changes require knowledge of the semantics of the types and their relationships, they most likely require more work from the maintainer to update the objects appropriately. However, evolution systems could provide a framework that a maintainer would be able to use to implement these evolution functions.

References

[Bar91] G. Barbedette. Schema modifications in the LISPO$_2$ persistent object-oriented language. In P. America, editor, *Proceedings of ECOOP '91, the Fifth European Conference on Object-Oriented Programming*, number 512 in Lecture Notes in Computer Science, Geneva, Switzerland, July 1991. Springer-Verlag.

[BKKK87] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the ACM SIGMOD 1987 Annual Conference*, pages 311–322, San Francisco, May 1987.

[Cas90] Eduardo Casais. Managing class evolution in object-oriented systems. In Dennis Tsichritzis, editor, *Object Management*, pages 133–195. Université de Genève, Switzerland, 1990.

[Cla92] Stewart M. Clamen. Type evolution and instance adaptation. Technical Report CMU-CS-92-133, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1992.

[LH90] Barbara Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. In *Proceedings of the Joint ACM OOPSLA/ECOOP '90 Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 67–76, Ottawa, Canada, October 1990.

[Mey90] Bertrand Meyer. Lessons from the design of the Eiffel libraries. *Communications of the ACM*, 33(9):68–88, September 1990.

[PS87] D. Jason Penney and Jacob Stein. Class modification in the GemStone object-oriented DBMS. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 111–117, Orlando, Florida, October 1987.

[SZ87] Andrea H. Skarra and Stanley B. Zdonik. Type evolution in an object-oriented database. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, Computer Systems Series, pages 393–413. The MIT Press, Cambridge, Massachusetts, 1987.

[Zdo90] S. Zdonik. Object-oriented type evolution. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, pages 277–288. ACM Press, New York, NY, 1990.