Should Superclasses be Abstract?

Walter L. Hürsch

College of Computer Science, Northeastern University
360 Huntington Avenue, Boston, MA 02115, USA
e-mail: huersch@ccs.neu.edu
phone: +1-617-373-3824 fax: +1-617-373-5121

Abstract. In object-oriented design and languages, abstractness of classes is a concept whose relationship to other concepts is not yet fully understood and agreed upon. This paper clarifies the concept of abstractness and examines the relationship between abstractness and inheritance. It does this by discussing several aspects of the so-called abstract superclass rule, a design rule for object-oriented programming which requires that all superclasses be abstract. In the course of this discussion, we evaluate in which situations this rule should be considered beneficial.

Keywords: Class Management, Object-Oriented Programming and Data Modeling, Software Evolution

1 Introduction

Object-oriented languages view inheritance and abstractness as two independent concepts. **Inheritance** is a relationship between a general class, called the superclass, and a specialized class, called the subclass. As a relationship between classes, inheritance does not single out specific classes; that is, any class can possibly be a superclass or a subclass. **Abstractness** is the inability of a class to create instances (objects) of itself. A class that has the ability to create instances is referred to as *instantiable* or *concrete*, otherwise it is called *abstract*. So the independence of inheritance and abstractness means that a class is instantiable *independently* of whether it is a superclass or a leaf class in the inheritance hierarchy.

This paper presents and evaluates a simple rule, called the abstract superclass rule (ASR), which removes the mutual independence of inheritance and abstractness. Unlike existing object-oriented languages, this rule requires that all superclasses be abstract.

Abstract Superclass Rule: All superclasses must be abstract.

The abstract superclass rule is deliberately formulated in a somewhat pointed way. However, rather than dogmatically putting it forward, we use it merely as a vehicle to discuss and clarify the issue of abstractness in object-oriented programming, and provide a collection of arguments for and against the rule. *In this sense, the abstract superclass rule should be considered as a guideline* rather than a strict rule. The designer of an application will have to apply the rule judiciously with careful consideration of the advantages and disadvantages for the particular situation.

The goal of this paper is to provide the designer with the basis to make an informed decision as to when to apply the rule successfully. For that purpose we collect arguments from a broad variety of software engineering perspectives. We will specifically answer the following questions:

1. Are there certain situations that prohibit the use of the ASR at all?
2. Is the ASR in any way unnecessarily restrictive?
3. What are the benefits or problems of the ASR with respect to data modeling?
4. How do systems designed with the ASR behave during software evolution?
5. How do systems designed with the ASR lend themselves to software reuse?
6. How does the ASR affect object-oriented programming?

In order to answer the above questions we will first analyze how the mechanism of abstractness is implemented in today's object-oriented languages. Then, some general aspects of the ASR are illuminated and clarified. It is shown that complying with the rule does not reduce the expressiveness of an object-oriented design. Also, we do away with a misconception of the ASR in connection with the covariance typing rule. Dodani et al. claim that their Abstract Concrete Type System (ACTS), a variant of the abstract superclass rule, allows covariant and contravariant typing rules to coexist in a single, type-safe environment [DT92]. We will show that this is a misconception, and that, in fact, the abstract superclass rule avoids run-time errors only in certain cases.

Data modeling is one of the essential areas in which a rule like the ASR needs to prove effective to be accepted. We will see that the abstract superclass rule resolves a number of problems when several notions of inheritance [BI94] coexist in the same program using one inheritance mechanism. Also, the rule appears naturally in the case for multiple inheritance, as discussed by Waldo [Wal91].

From the software evolution perspective, the ASR becomes problematic with respect to subclassing. Unless leaf classes are essentially split into an abstract class for subclassing and a concrete class for instantiation, the rule is not truly useful. The splitting in turn results in a proliferation of classes placing an additional burden on maintenance. However, the ASR allows the representation of some objects to be changed without affecting the representation of other objects. Moreover, the addition or deletion of properties to objects of one class don't necessarily propagate to the objects of other classes.

For the reusability aspect, the same caveat as above applies in terms of subclassing. On the other hand, the benefits of the abstract superclass rule for object-oriented frameworks and libraries, the major building blocks of reusability, have been widely observed before [Deu83, JF88, Joh93].

The paper is organized as follows. Section 2 reviews and discusses the definition and mechanism of abstractness for the three languages Smalltalk, C++, and Eiffel. As part of this, we propose the de-coupling of the concept of class abstractness from the presence of abstract features (e.g., pure virtual functions in

C++). Section 3 then answers the above set of questions by discussing the ASR from several vantage points of software engineering: data modeling, software evolution and reusability. In addition, issues related to covariance, programming, and simplicity are covered. The final section gives a summary of the pros and cons, and an evaluation of the rule for designers and programmers.

2 Abstractness in Object-Oriented Languages

Before discussing the abstract superclass rule in detail, let us review the mechanism of abstractness for the three object-oriented languages Smalltalk, C++ and Eiffel. As CLOS [Kee89] does not know the notion of abstract classes nor the notion of abstract methods we will not take it into further consideration here.

In Smalltalk [GR83], an abstract class is described and defined as follows.

> "Abstract superclasses are created when two classes share a part of their descriptions and yet neither one is properly a subclass of the other. A mutual superclass is created for the two classes which contains their shared aspects. This type of superclass is called *abstract* because it was not created in order to have instances."
> "Abstract class: A class that specifies protocol, but is not able to fully implement it; by convention, instances are not created of this kind of class."

The corresponding definition in C++ [ES90] is:

> "The abstract class mechanism supports the notion of a general concept of which only more concrete variants can actually be used."
> "An abstract class is a class that can be used only as a base class of some other class; no objects of an abstract class may be created except as objects representing a base class of a class derived from it. A class is abstract if it has at least one pure virtual function."

The corresponding definition in Eiffel [Mey92, Mey88] is:

> "A class which has at least one deferred feature is itself said to be **deferred**; a non-deferred class is called an effective class. A feature is made deferred by declaring it without choosing an implementation."
> "Deferred class no-instantiation rule: *Create* may not be applied to an entity whose type is given by a deferred class."

Syntactically, an abstract class is expressed (1) in Smalltalk: simply by not specifying an implementation for at least one declared method, (2) in C++: by declaring at least one method as pure virtual ("`= 0`"), and (3) in Eiffel: by declaring the class and at least one method as "deferred". Note that in C++, unlike in Smalltalk and Eiffel, an abstract method can still have an implementation. The only way to designate such a method as abstract is by syntactically specifying abstractness through the use of "`= 0`". A summary of the above abstractness mechanisms is given in Table 1.

Language	Terminology		Syntax		Implementation
	class	method	class	method	of abstract method
Smalltalk	abstract	abstract	–	–[a]	not possible[b]
C++	abstract	pure virtual	–	= 0	possible[c]
Eiffel	deferred	deferred	`deferred`[d]	`deferred`	not possible[e]

[a] no syntactic means
[b] by definition
[c] can only be called through scope resolution operator (::)
[d] needs to be coupled with at least one deferred method
[e] checked by compiler

Table 1. Summary of existing abstractness mechanisms

There are two problems with the above definitions. First, a closer examination reveals that there are two concepts involved in abstractness: ability to create instances, and presence of abstract methods. The fact that these two different concepts are merged into one single concept precludes the designer from using them separately. Of course, if a class contains abstract methods then the class cannot be safely instantiated, because if an instance were created, it would not be able to successfully respond to all its messages. So the presence of an abstract method implies the inability to instantiate objects. However, the converse does not hold: a class might not be intended for instantiation and yet might have no abstract methods.

In the case of the above languages, if a class had a priory no abstract methods, it would not be possible for the designer to make the class abstract without defining a dummy undefined method for it[1]. This is unsatisfactory since one could well imagine cases where a class is designed to be 'abstract' but does provide implementations for all its methods. (An example of this is easily found with data inheritance as discussed in section 3.3.)

There is a second problem to the discussed abstractness mechanisms. To find out whether a given class is abstract, a programmer has to inspect all methods of that class and check whether at least one of them is abstract. Even worse, abstract methods are inherited as abstract methods which forces one to check also all superclasses of the class for abstract methods. For large class hierarchies, this is obviously a rather tedious task. In Eiffel, this problem is somewhat alleviated by the use of the keyword `deferred` preceding the class definition. However, Eiffel still requires to define at least one deferred method for a deferred class resulting in the same problem when determining whether a class needs to be declared deferred or not.

To remedy the above problems we propose the following definition of abstractness. We regard the ability to instantiate objects as the more profound

[1] It has been suggested for C++ to provide a protected constructor instead of a dummy pure virtual function to make a class abstract. However, this has the flaw that friends and subclasses are still able to instantiate the class.

concept behind abstractness. Therefore, we define abstractness of a class as *the inability to instantiate objects.* Note that this definition deliberately does not make any reference to, and thus is independent of, the presence of abstract methods. Syntactically, we propose to actively specify the abstractness of a class through a specific keyword (e.g., `abstract`, or `deferred`). This allows the designer to clearly convey a class's intention in its definition and solves the second problem discussed above.

3 Discussion of the Abstract Superclass Rule

Now that the notion of abstractness is clarified, we are ready to discuss the impact of abstractness on the design of object-oriented applications. The abstract superclass rule is used as a means to do this. This section discusses the abstract superclass rule by viewing it from various aspects of software engineering, thus collecting arguments for and against it from different vantage points.

An immediate objection to the abstract superclass rule may be raised because of its automatic coupling of the two a priori independent concepts abstractness and "superclass-ness". This problem is similar to the problem of coupling the presence of abstract methods with the ability to instantiate objects as we have seen it in section 2. Such a coupling of one independent class concept to another is never desirable since often the designer wants to use one concept but not the other, which is not possible if the two are coupled.

The above argument would not hold, however, if through the discussion of the following aspects of the abstract superclass rule, it turned out that the rule should actually be followed in all situations. In that case, abstractness and superclass-ness *need* be coupled and the above deficiency would in fact be an essential feature of the rule. As we will see, however, the rule does not hold in the necessary strictness, and so the above coupling is, indeed, generally undesirable.

3.1 The Expressiveness Aspect

First we show that the abstract superclass rule does not reduce the expressiveness of the object-oriented design. In particular, any class structure that contains an instantiable superclass can be transformed into an equivalent structure that conforms to the abstract superclass rule.

The transformation creates an additional, empty subclass of the instantiable superclass. "Empty" means that the subclass inherits all its data and behavior from its superclass. The purpose of the old superclass is still to serve for subclassing, but it can now become abstract since the new subclass takes over its place for instantiation purposes.

The described transformation is illustrated in Fig. 1. Consider the class structure on the left. Class `A` is instantiable and has two subclasses `B` and `C`. This class structure can be transformed into one that complies with the abstract superclass rule on the right. The transformation provides an extra concrete subclass of `A`, say `A'`, which inherits all of its data and behavior from `A`. `A` becomes abstract, and both B and C still inherit from A. In the transformed hierarchy, the purpose of class A' is to instantiate "A"-objects while the purpose of class A is to serve as a superclass.

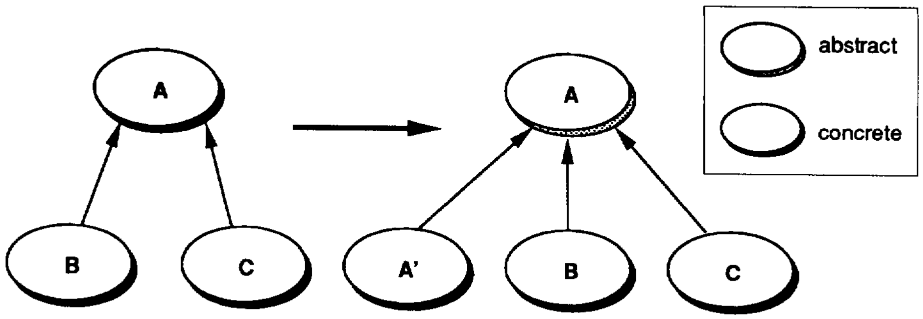

Fig. 1. Class hierarchy transformation to comply with the abstract superclass rule

The above transformation of hierarchies is effective and straightforward. But even if there is existing code, the impact of the transformation is manageable, albeit not as effortlessly. For C++ and Eiffel, it is advantageous if the old superclass keeps its name and the new concrete subclass gets a new name as indicated in the example. In this case, constructor calls and type declarations of the superclass (except for pointers and references to the superclass) need be updated. For C++, also special member functions like constructors and assignment operators need be provided for A' since they are not inherited. For Smalltalk, the only references to the type name occur in constructor calls and superclass declarations. Therefore, it is advantageous if the old class name is transferred to the new concrete class. Then, the only point where we need to change the code is in the superclass declarations of the immediate subclasses, which, in some cases, is done automatically by the environment.

Obviously, the above transformation is purely mechanical and the transformed class hierarchy is in no essential way "better" than the original. This is analogous to the classic Böhm-Jacopini construction [BJ66] which showed that any program structure using jump (goto) statements can be mechanically converted to an equivalent program using only structured-programming constructs. The result of a Böhm-Jacopini transformation is also not essentially "better" than the original. "Proving" that structured programming is better was a totally different task. Similarly, the above transformation of class hierarchies merely proves the equivalency of the two designs. Showing the advantages and/or disadvantages of the abstract superclass rule is another matter and will be dealt with in the subsequent sections.

In conclusion, the abstract superclass rule does not reduce any expressiveness in the design of an object-oriented application. However, for existing applications, the benefits of the rule need to outweigh the efforts necessary to transform existing code.

3.2 The Covariance Aspect

It has been widely recognized that the covariant rule on method redefinition during inheritance, as used for example in Eiffel [Mey88, Mey92], can lead to run-time errors and thus is not type-safe [CCHO89, Coo89, DT92]. It has also been claimed that a strict use of the abstract superclass rule makes the covariant method redefinition rule type-safe [DT92]. We will show that this is a misconception and that the abstract superclass rule does not make programs that use the covariant rule type-safe.

A method in a subclass is redefined according to the covariant redefinition rule if both result type and argument types of the redefined method are more specific than the result type and argument types of the original method in the superclass. In the following, we will call a program that employes the covariant rule on method redefinition during inheritance simply a *covariant program*. A typical situation where a covariant program can lead to a runtime error occurs when a piece of code expects an object of a subclass but actually deals with an object of a superclass. Then the object could possibly be sent a message which is undefined in the superclass. The problem is usually depicted through the following example taken from [DT92].

```
 1  Point
 2    x: Int;
 3    y: Int;
 4    eq(p:Point): Boolean
 5      = ( x==p.getx() AND y==p.gety() )
 6
 7  ColorPoint inherits Point
 8    c: Color;
 9    eq(p:ColorPoint): Boolean
10      = (x==p.getx() AND y==p.gety() AND c.coloreq(p.getc()))
11
12  p1,p2: Point;
13  cp: ColorPoint;
14  p1 := new Point(10,20);
15  cp := new ColorPoint(10,20,red);
16  p2 := cp;
17
18  p2.eq(p1);  -- run-time failure, would be avoided by ASR
```

We assume that accessors like `getx()` and `gety()` are suitably defined or automatically created elsewhere. The program fragment satisfies the covariant typing rule, but will eventually break at run-time in the call to the method `eq` on line 18. The dynamic binding mechanism will execute the `eq` code of the subclass `ColorPoint` with an actual argument object of class `Point` which does *not* have a color attribute and thus will issue a "message not understood" error when that attribute is tried to be accessed.

In the above example, the abstract superclass rule would avoid this error since it does not allow objects of superclasses to be instantiated; that is, it would not be possible in the first place to pass in a "wrong" object of class `Point` in a call to the `eq` method of class `ColorPoint`.

Could it be that the abstract superclass rule makes *all* covariant programs type-safe? Dodani and Tsai actually claim it does [DT92]. In their Abstract Concrete Type System (ACTS) they claim to provide a uniform solution to the problem of developing a type-safe hybrid type system capable of handling both covariance and contravariance.

A class hierarchy built with the ACTS is a little less restricted than a hierarchy built with both the abstract superclass rule and the covariant typing rule. An ACTS hierarchy is two-tiered where abstract classes are interior nodes of the hierarchy while concrete classes form the leaf nodes. Abstract classes appearing as leaves need to be concretized before they can be used. Any subtree rooted by a concrete class must have all nodes as concrete classes as well. In addition, the ACTS prescribes the contravariant typing rule on concrete-to-concrete inheritance while the covariant typing rule is prescribed for both the abstract-to-abstract and abstract-to-concrete inheritance relationships. For our discussion, only the last two prescriptions are of importance since concrete-to-concrete inheritance is forbidden by the abstract superclass rule.

Thus, hierarchies built with the abstract superclass rule and the covariant typing rule are a special case of the ACTS. The difference is that in a class hierarchy built with the abstract superclass rule the "leaf" layer of concrete classes has only depth 1 while in the ACTS it can be arbitrarily deep as long as all classes in the layer are concrete. In what follows we will show that the ACTS separation of abstract and concrete classes in the class hierarchy is in fact *not* sufficient to make the covariance rule type-safe.

The proof consists of a program for which we show that (1) it follows the abstract superclass rule, (2) it is type-correct under the covariant typing rule, and (3) yields a "message not understood" error at run-time. Due to limited space, we cannot provide a full formal analysis of the program under the ACTS type checking algorithm. In addition, the original presentation of ACTS is flawed and somewhat imprecise [Tsa94]. We therefore appeal to the intuition of the reader and refer to [Hür94] for full details.

The counter-example below is similar to the previous example, but adds another subclass, `ThreeDPoint`, to the class `Point`. `ThreeDPoint` contains a third attribute for the z-axis of a point and redefines the method `eq`.

The program fragment satisfies the covariant typing rule, but will also break at run-time in the calls to the method `eq` on lines 25–26. Both calls break for the same reason: the dynamic binding mechanism will execute the `eq` code of one subclass (e.g., `ColorPoint`) with an actual argument object of its sibling class (e.g., `ThreeDPoint`) which does *not* have all of the required attributes (e.g., `c`) and thus will issue a "message not understood" error when that attribute is accessed. Note that this error occurs irrespective of whether class **Point** is abstract or not.

```
 1  Point   -- abstract
 2    x: Int;
 3    y: Int;
 4    eq(p:Point): Boolean
 5      = ( x==p.getx() AND y==p.gety() )
 6
 7  ColorPoint inherits Point
 8    c: Color;
 9    eq(p:ColorPoint): Boolean
10      = (x==p.getx() AND y==p.gety() AND c.coloreq(p.getc()))
11
12  ThreeDPoint inherits Point
13    z: Int;
14    eq(p:ThreeDPoint): Boolean
15      = ( x==p.getx() AND y==p.gety() AND z==p.getz() )
16
17  p1,p2: Point;
18  cp: ColorPoint;
19  tp: ThreeDPoint;
20  cp := new ColorPoint(10,20,red);
21  tp := new ThreeDPoint(10,20,30);
22  p1 := cp;
23  p2 := tp;
24
25  p1.eq(tp);  -- run-time failure, would NOT be avoided by ASR
26  p2.eq(cp);  -- run-time failure, would NOT be avoided by ASR
```

Another way to look at the problem is depicted in Fig. 2. Consider the call `p1.eq(tp)` in line 25. Since `p1`'s value is an object of class `ColorPoint`, the dynamic binding mechanism executes the code of method `eq` attached to `ColorPoint` which assumes the argument is of class `ColorPoint` as well. However, the type system assumes that the argument is an object of type `Point` or one of its subclasses. This results in four different kinds of type regions for the actual arguments of the method call: (1) All objects of subclasses of `ColorPoint` are safe (lightly shaded region). (2) Objects of class `Point` are unsafe but disallowed by the abstract superclass rule. (3) Objects of sibling classes of `ColorPoint` (like `ThreeDPoint`, darkly shaded region) are unsafe but allowed by the ACTS. (4) Objects of superclasses of `Point` and other classes are unsafe and generally disallowed. Thus the abstract superclass rule, in effect, prohibits only a comparatively small set of objects from being passed to the method call.

In conclusion, we have shown that neither the ACTS nor the abstract superclass rule eliminates run-time errors in the presence of covariance. In particular, the abstract superclass rule does *not* solve the problem of type-safeness for the

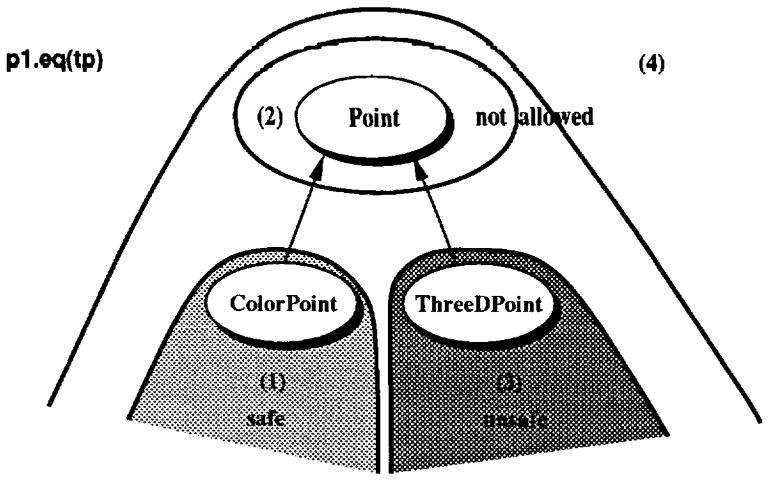

Fig. 2. The regions of type-(un)safety

covariant typing rule[2]. However, it does help to avoid some of the errors in the presence of covariance. Note that this result is not an argument *against* the abstract superclass rule, it just rules out a seeming advantage of the rule.

3.3 The Data Modeling Aspect

For object-oriented data modeling there are three important factors for which the abstract superclass rule makes a difference. All of them have to do with the confusion of what the notion of inheritance really means and when it should be applied.

The notion of inheritance As we know, the key concept of object-oriented programming and one of the major means to model real world domains in the object paradigm is the notion of inheritance. As such, *some* inheritance mechanism is provided and even required in every object-oriented programming language (OOPL). While for a given OOPL it is clear *how* to use its inheritance mechanism, the question of *when* to apply this mechanism to a real world situation is usually less clear. In fact, there has been quite some debate over what the notion of inheritance means; that is, what real world situations it should and can model [Bra83, HO87, WZ88, LP91, Win92, BI94].

For example, Winkler [Win92] contrasts the "concept-oriented view" (COV) of inheritance with the "program-oriented view" (POV). The COV uses inheritance for classifications and characterizes the typical inheritance relationship between a subclass `C` and its superclass `B` as "`C` *is-a* `B`". The POV views inheritance merely as a technical concept of software technology that defines a

[2] For a proposal of how to make Eiffel type-safe, see [Coo89]

monotonic extension relation between the superclass and its subclass. (Wegner [Weg90] uses the phrases "logical hierarchy" for COV, and "physical hierarchy" for POV; other commonly used terms are "interface inheritance" and "implementation inheritance", respectively.)

Another vantage point of the same issue is given by Brachman [Bra83] who shows that there are six kinds of generic-generic relations and four different kinds of generic-individual relations for semantic networks, all grouped together under one label "IS-A". Baclawski et al. [BI94] summarize the situation as follows: "there are no 'standard' conceptual hierarchies." They point out that "How inheritance is to be incorporated in a specific system is up to the designers of the system, and it constitutes a policy decision that must be implemented with the available mechanisms." In essence, there is not one single *policy* for how to use the *mechanism* inheritance.

A serious problem occurs when two policies cannot coexist using the same inheritance mechanism. This is the case for the two major policies COV and POV, as discussed by Winkler [Win92]. Winkler finds that in certain situations the COV may result in programs which are "awkward, inefficient, and even incorrect." For example, in COV, squares are modeled as a subclass of rectangles since one could say "a square *is-a* rectangle with all four sides equal." However, a class hierarchy in which `Square` *is-a* `Rectangle` results in three problems: (1) Each `Square` object contains *two* data components for the side lengths, where *one* would be sufficient. (2) `Square` inherits methods from `Rectangle` that are not applicable to a square (e.g., `setHeight()`). (3) In order to enforce the semantics of a square, the preconditions of some inherited methods must be strengthened (e.g., `setSides()`), invalidating the substitutability of rectangles with squares.

How does this relate to the abstract superclass rule? The interesting point is that the abstract superclass rule lets the two policies COV and POV coexist using the same inheritance mechanism without exhibiting the problems presented by Winkler. This usage of the rule has been successfully employed in the Demeter data model [LX93, SLHS94]. Grosberg proposes the same solution in response to Winkler's article [Gro93]. As he points out, the designer needs to distinguish between the particular and the general by not implementing both in the same class. In the case of the square–rectangle hierarchy, the solution he outlines is the hierarchy depicted on the right-hand side of Fig. 3.

Note that the above transformation exactly follows the abstract superclass rule transformation for producing equivalent hierarchies. As Grosberg observes, the above solution remedies all the deficiencies brought up by Winkler: substitutability of squares is still preserved, squares don't inherit extraneous methods, and there is no need for inappropriate preconditions. Moreover, polymorphism is still supported for the `Abst_Rect` class.

Multiple inheritance There is another interesting case in which two other policies of using the inheritance mechanism make extensive use of the abstract superclass rule. These two policies were first introduced by Waldo [Wal91] in his "case for multiple inheritance". He termed them *interface inheritance* and *data inheritance.*

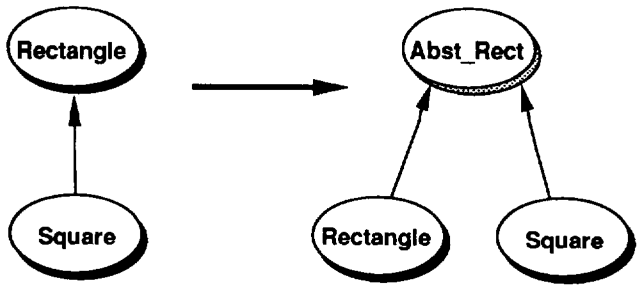

Fig. 3. Class hierarchy transformation for the square–rectangle example

Interface inheritance occurs when the only thing a class inherits is a set of interfaces defined in the superclass. In particular, the superclass does not provide any implementation nor does it define any data representation, which effectively makes it an abstract superclass. Essentially, the superclass serves as a repository of common method interfaces for all of its subclasses.

Data inheritance is the complementary notion of interface inheritance; it occurs when the only thing a class inherits is a set of instance variables. Specifically, the superclass does not provide any methods with the data. The purpose of the superclass is thus to serve as a repository of common instance variables for its subclasses *without* imposing any behavior. Note that, although there are no abstract methods present in such a superclass, the implied intention of the class strongly suggests it be abstract since a behaviorless object of the class would hardly make any sense.

The reason Waldo discussed these two sorts of inheritance was because they exhibit the need for multiple inheritance (for more details, refer to [Wal91] and [Sak92]). For our discussion, their importance lies in the fact that they are a good example for when to use the abstract superclass rule.

Classes as collection of objects There is yet another situation where the abstract superclass rule turns out to be useful for data modeling. Consider again the class hierarchy on the left-hand side of Fig. 1 with all classes being concrete. Assume that for some reason we need a variable whose values can only consist of objects of class A *excluding* objects of classes B and C. Such a variable is not possible for the given hierarchy since any variable declared to be of class A can automatically also contain objects of classes B and C.

Contrast this to the situation where the class hierarchy conforms to the abstract superclass rule (right-hand side of Fig. 1). Here, any set of objects, A's, B's, C's, or a heterogeneous collection of all three of them can be the domain of a variable. Thus the abstract superclass rule separates the dual responsibility of a class to stand for both a collection of its objects only, and a collection of its objects and all its subclasses' objects.

Conclusion We have analyzed three situation where the abstract superclass rule turned out to be useful. First, following the abstract superclass rule allows the designer to employ both the COV and the POV policies within the same inheritance mechanism. Second, for multiple inheritance both interface and data inheritance necessitate superclasses to be abstract. Third, the abstract superclass rule releaves classes from their dual responsibility at the object level to stand for two different collections of classes. The abstract superclass rule can thus be considered beneficial for object-oriented data modeling.

3.4 The Evolution Aspect

So far we have considered the abstract superclass rule only from a static vantage point. But software systems are dynamic. As described by the spiral life cycle model [Boe86], any successful software system will eventually be changed, be it to improve, update, adapt, or strengthen it. What are the consequences of employing the abstract superclass rule in an environment that evolves? We will consider two evolutionary aspects: subclassing from existing classes, and modification of class properties.

Subclassing Assume the class hierarchy conforms to the abstract superclass rule and one needs to build a subclass from an existing class C in the hierarchy. If C is already a superclass or abstract then there is no problem. However, if C is not abstract it becomes a concrete superclass through the subclassing process violating the abstract superclass rule. One needs then to transform the class structure as indicated earlier in order to comply again to the rule. Clearly, this need for transformation is unsatisfactory and defeats one purpose of inheritance, namely ease of code reuse. The situation is aggravated when C and the original class hierarchy reside in a library, where the source code is not accessible to the programmer. In this case, the above transformation cannot even be performed and there is no choice but to violate the rule.

Even if the application does not use libraries, the designer of the class hierarchy needs to make a decision how to prepare for evolution in the presence of the abstract superclass rule. To avoid later restructuring of the hierarchy the guidelines below need be followed by both libraries and regular applications. For each instantiable (leaf) class, depending on the expected later use of the class, do the following. (1) If the class is to serve for instantiation only, provide a concrete class. (2) If the class is to serve for subclassing only, provide an abstract class. (3) If the class is to serve for both subclassing and instantiation, provide both an abstract superclass and a concrete subclass. (4) If the expected use of a class cannot be predetermined then it is safer to provide both an abstract and a concrete class.

The guidelines are especially important when designing a library or a framework with the abstract superclass rule due to the fact that a later user of the library generally cannot change it. Therefore, we will call these guidelines collectively the *abstract library rule* (see Table 2). Following the abstract library rule

Expected later use	Implementation
instantiation only	concrete class
subclassing only	abstract class
instantiation and subclassing	two classes: abstract superclass and concrete subclass
unknown	two classes: abstract superclass and concrete subclass

Table 2. The abstract library rule for leaf classes

guarantees that classes can be subclassed without restructuring the hierarchy to maintain the abstract superclass rule. In a sense, the abstract library rule is a necessary consequence of the abstract superclass rule.

What is the impact of following the abstract library rule for an application? On the one hand, quite a serious disadvantage of the rule is that it results in a proliferation of classes. In the worst case, it doubles the number of classes of an application, namely when the designer needs to follow guideline (3) for every class in the application. For large hierarchies with a lot of leaf classes, this turns into a maintenance nightmare. But not only that, half of the classes are simply clones of their superclasses and don't add any data or behavior, making them effectively "useless". A similar observation has been made by Riel [Rie94] who proposes a "specialization pattern" heuristic to remove the abstract superclass in this situation.

On the other hand, a slight benefit of the abstract library rule is that it increases the expressiveness or granularity of the import and export facilities of a library. For instance, an application that imports only the concrete class but not the abstract counterpart conveys the fact that no new subclasses will be used in the application. The rule basically splits the interface of a library into two types of classes: those provided for subclassing and those provided for instantiation.

Summarizing, the abstract superclass rule cannot be enforced in applications that use external libraries. The rule does not allow "unplanned" subclassing and needs to be paired with the abstract library rule if later subclassing should be possible without restructuring the hierarchy. The abstract library rule in turn results in a proliferation of classes increasing the burden of maintenance.

Addition/deletion of class properties In the second evolutionary aspect, assume one wants to add or delete properties from objects of an instantiable superclass. This is not possible without affecting, at the same time, all instances of all the subclasses down the hierarchy, a formidable trickle-down effect in some cases. For example, assume one wants to add an attribute to all objects of class `Rectangle` in Fig. 3. In the class hierarchy on the left-hand side, this is not possible without changing at the same time all objects of subclasses of `Rectangle` (e.g., objects of class `Square`).

If the class hierarchy was built with the concept-oriented view (COV) of inheritance (see section 3.3) then the instantiable superclass and its subclasses are conceptually related by the *is-a* relation and any change to the superclass *should* actually be propagated to instances of subclasses. In other words, the trickle-down effect is desired. However, when the class hierarchy was built with the program-oriented view (POV) of inheritance then the superclass and its subclasses are generally not conceptually related and a change in the superclass should usually *not* be passed to instances of subclasses. Thus, the trickle-down effect is not desired.

The advantage of the abstract superclass rule is that all objects are instances of leaf classes and hence no change propagation to objects of subclasses takes place. With the rule it is possible to add parts to or delete direct parts from existing objects at any time in the evolution process without affecting other objects. When deleting inherited parts the class must become a subclass of a different superclass depending on which property was deleted.

Change of representation Johnson and Foote [JF88] point out another benefit of the abstract superclass rule for the evolution aspect. They claim that in general, it is better to inherit from an abstract class than from a concrete class. The reason is that abstract classes generally do not have to provide their own data representation, and so future concrete subclasses can use their own representation without the danger of conflicts.

To guarantee that the representation of a class can be easily modified, two other guidelines need be followed in addition to the abstract superclass rule. First, all instance variables may only be accessed through accessor methods. This makes the way the representation is accessed independent from the actual representation. Second, only leaf classes my actually define instance variables. In other words, the actual representation is chosen by each instantiable class individually while abstract superclasses only provide behavior and a common interface.

Conclusion On the one hand, we have seen that the abstract superclass rule becomes problematic in the presence of legacy code and external libraries. To make the rule effective and safe for later subclassing, it needs to be paired with the abstract library rule which results in a proliferation of classes. On the other hand, the abstract superclass rule is beneficial for programmers working with the program-oriented view of inheritance, and it allows for easier change of representation.

3.5 The Reusability Aspect

In object-oriented systems, libraries and frameworks are the major building blocks of reuse. We will therefore primarily consider the impact of the abstract superclass rule on object-oriented libraries and frameworks. The higher quality of a framework or library induced by the rule implies a higher likelihood of reusability.

As discussed in the previous section, one problem of the rule is with respect to subclassing. Libraries need to be built with the abstract library rule if they want to be truly reusable. However, in many cases a library or framework provide mostly abstract classes anyway because the classes need be subclassed for actual use in an application. In that case, no proliferation of "useless" classes occurs.

In support of the abstract superclass rule, Johnson and Foote [JF88, Rule 6] require that the top of the class hierarchy should be an abstract class. In addition, from his experience with object-oriented frameworks, Johnson [Joh93] points out that classes provided in object-oriented libraries should always be abstract.

The abstract library rule seems to be very natural for object-oriented frameworks. Deutsch [Deu83] defines a *framework* to be an abstract object-oriented design consisting of an abstract class for each major component. The simplest example of a framework is a so-called single-class framework constisting of nothing else but a single abstract superclass [Deu89].

As a matter of fact, it is their abstractness which makes frameworks useful for a wide range of applications in a specific domain since it allows them to solve a problem at a higher level of abstraction without knowing all the specific implementation details. Johnson and Foote [JF88] find that frameworks usually contain abstract classes and that frameworks can even be built on top of other frameworks by sharing some of these abstract classes.

In addition to being "abstract" in the technical sense, abstract classes usually are also at a higher level of abstraction from an application point of view. This means that an abstract class represents a generalization of the concept it is used for in the current application and therefore is a likely candidate of reuse in later applications.

3.6 The Simplicity Aspect

We have already seen that the abstract superclass rule can simplify object-oriented programming in a number of ways. First, by designing a system in compliance with the abstract superclass rule, it is easier for the programmer to find out which classes can be instantiated and which cannot. Second, adding or deleting attributes from objects as well as changing their representation is facilitated if the abstract superclass rule has been followed.

But the abstract superclass rule also introduces a certain uniformity into the way dynamic binding and polymorphism mechanisms work. As Wilde et al. point out [WMH93]: "For the maintainer, dynamic binding and polymorphism are two-edged swords. They give programs the flexibility that is a main objective of object-oriented programming, but they also make programs harder to understand." For example, a message dispatch through a variable of a superclass can end up executing different code depending on the actual object referred to by the variable. Contrast the ways to find out which method is actually executed. The search for the actual method starts from the class of the actual object proceedings up the hierarchy. In a non-ASR hierarchy, the search can possibly start anywhere within the hierarchy. In an ASR hierarchy, the search always starts from the bottom of the hierarchy since the actual object can only be an instance of a leaf class. This unity of mechanism reduces complexity in deep hierarchies.

4 Conclusion

There is not one single reason why the abstract superclass rule is good or bad. It's value can only be determined by collecting a number of small arguments from various aspects of software engineering. The conclusions found in these aspects are summarized in Table 3.

Aspect	Finding	Benefit
Concept Coupling	ASR couples abstractness and "superclass-ness"	–?
Expressiveness	ASR does not reduce expressiveness	+/–
Covariance	ASR is *not* a remedy for covariance	+/–
Data Modeling	ASR allows several inheritance policies to coexist	+
Evolution	ASR needs to be paired with abstract library rule (proliferation of "useless" classes)	–
	ASR is favorable for object evolution	+
Reusability	ASR promotes certain reusability	+
Simplicity	ASR simplifies programming	+

Table 3. Summary: benefits of the abstract superclass rule

We have found that the abstract superclass rule does not reduce any expressiveness in the design of an object-oriented application. But for existing applications, the benefits of the ASR need be contrasted with the effort of code transformation which are not negligible for typed languages. The ASR has been wrongly thought to render programs type-safe which use the covariant method redefinition rule during inheritance. The fact that it does not is neither a benefit nor a disadvantage of the rule.

We saw that the ASR resolves a number of problems when several notions of inheritance coexist in the same program using one inheritance mechanism. In addition, many forms of multiple inheritance make ample use of the rule.

The ASR essentially inhibits subclassing from instantiable classes. This destroys one of the major benefits of object-oriented programming: flexibility and reusability. Unless leaf classes are essentially split into an abstract class for subclassing and a concrete class for instantiation, the rule is not truly useful. But even then, the proliferation of classes due to the splitting places a burden on maintenance and readability of the design. On the other hand, the ASR allows the representation of some objects to be changed without affecting the representation of other objects. Moreover, the addition or deletion of properties to objects of one class does not necessarily propagate to the objects of other classes.

With respect to reusability, the same caveat as above applies in terms of subclassing. But the ASR is generally considered beneficial for object-oriented frameworks and libraries.

Summarizing, we can conclude that the abstract superclass rule is in many situations beneficial and can generally be considered a good guideline for object-oriented design and programming. Some caveats apply however, the reasons we found for enforcing the rule were by no means compelling. As for any rule, exceptions need be allowed in certain situations. The designer of an application will have to apply the rule judiciously with careful consideration of the advantages and disadvantages for the particular situation. In this sense, the abstract superclass rule should be viewed as a rule of thumb, rather than a strict rule.

Acknowlegdments
Many useful comments on the topic and on earlier drafts of this paper were received by Ken Baclawski, Ralph Johnson, Karl Lieberherr, Cristina Lopes, Ignacio Silva-Lepe, Cun Xiao, and the anonymous reviewers. The author thanks especially Jens Palsberg and Markku Sakkinen for many inspiring discussions. This work has been generously supported by Mettler-Toledo AG, Switzerland.

References

[BI94] Kenneth Baclawski and Bipin Indurkhya. The notion of inheritance in object-oriented programming. *Communications of the ACM*, 36, 1994. Technical correspondence; accepted for publication.

[BJ66] Corrado Böhm and Giuseppe Jacopini. Flow Diagrams, Turing Machines and Languages with only Two Formation Rules. *Communications of the ACM*, 9(5):366–371, May 1966.

[Boe86] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *Software Engineering Notes*, 11(4), 1986.

[Bra83] Ronald J. Brachman. What IS-A Is and Isn't: An Analysis of Taxonomic Link in Semantic Networks. *IEEE Computer Magazine*, 16(10):30–36, October 1983.

[CCHO89] Peter S. Canning, William R. Cook, Walter L. Hill, and Walter G. Olthoff. Interfaces for strongly-typed object-oriented programming. In Norman Meyrowitz, editor, *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 457–467, New Orleans, Louisiana, October 1989. ACM Press. Special Issue of SIGPLAN Notices, Vol.24, No.10.

[Coo89] William R. Cook. A Proposal for Making Eiffel Type-Safe. *The Computer Journal*, 32(4):305–311, 1989. A preliminary version of this paper appeared in the proceedings of ECOOP '89.

[Deu83] L. Peter Deutsch. Reusability in the Smalltalk-80 Programming System. In *Proceedings of the Workshop on Reusability in Programming*, pages 72–76. ITT, 1983.

[Deu89] L. Peter Deutsch. Design Reuse and Frameworks in the Smalltalk-80 System. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability; Volume II, Applications and Experience*, chapter 3, pages 57–71. ACM Press, 1989. ISBN 0-201-50018-3.

[DT92] Mahesh Dodani and Chung-Shin Tsai. ACTS: A Type System for Object-Oriented Programming Based on Abstract and Concrete Classes. In

O. Lehrman Madsen, editor, *European Conference on Object-Oriented Programming*, pages 309–324, Utrecht, The Netherlands, June/July 1992. Springer Verlag, Lecture Notes in Computer Science. Vol. 615.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990. ISBN 0-201-51459-1.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80, the Language and its Implementation.* Addison-Wesley, Reading, MA, 1983. ISBN 0-201-11371-6.

[Gro93] John A. Grosberg. Comments on considering 'class' harmful. *Communications of the ACM*, 36(1):113–114, January 1993. Technical correspondence.

[HO87] Daniel C. Halbert and Patrick D. O'Brien. Using types and inheritance in object-oriented programming. *IEEE Software*, 4(5):71–79, September 1987.

[Hür94] Walter L. Hürsch. Covariance in the presence of the abstract superclass rule. Technical Report NU-CCS-94-04, College of Computer Science, Northeastern University, Boston, MA, March 1994.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, pages 22–35, June/July 1988.

[Joh93] Ralph Johnson. Abstract superclasses in object-oriented libraries. Private communication, November 1993.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Addison-Wesley, Reading, MA, 1989. ISBN 0-201-17589-4.

[LP91] Wilf LaLonde and John Pugh. Subclassing $\neq$ Subtyping $\neq$ Is-a. *Journal of Object-Oriented Programming*, 3(5):57–62, January 1991.

[LX93] Karl J. Lieberherr and Cun Xiao. Formal Foundations for Object-Oriented Data Modeling. *IEEE Transactions on Knowledge and Data Engineering*, 5(6), June 1993.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* International Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ, 1988. ISBN 0-13-629049-3.

[Mey92] Bertrand Meyer. *Eiffel: The Language.* Object-Oriented Series. Prentice-Hall, Englewood Cliffs, NJ, 1992. ISBN 0-13-247925-7.

[Rie94] Arthur J. Riel. Object-Oriented Design Heuristics: Gateways for Design Transformation Patterns. Submitted for publication, February 1994.

[Sak92] Markku Sakkinen. A Critique of the Inheritance Principles of C++. *Computing Systems, The Journal of the USENIX Association*, 5(1):69–110, Winter 1992.

[SLHS94] Ignacio Silva-Lepe, Walter L. Hürsch, and Greg Sullivan. A Report on Demeter/C++. *C++ Report*, 6(2):24–30, February 1994.

[Tsa94] Chung-Shin Tsai. Corrections to the Abstract Concrete Type System. Private communication, March 1994.

[Wal91] Jim Waldo. Controversy: The Case for Multiple Inheritance in C++. *Computing Systems, The Journal of the USENIX Association*, 4(2):157–171, 1991.

[Weg90] Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, January 1990.

[Win92] Jürgen F. H. Winkler. Objectivism: 'class' considered harmful. *Communications of the ACM*, 35(8):128–130, August 1992. Technical correspondence.

[WMH93] Norman Wilde, Paul Matthews, and Ross Huitt. Maintaining object-oriented software. *IEEE Software*, 10(1):75–80, January 1993.

[WZ88] Peter Wegner and Stanley B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In S. Gjessing and K. Nygaard, editors, *European Conference on Object-Oriented Programming*, pages 55–77, Oslo, Norway, August 1988. Springer Verlag, Lecture Notes in Computer Science.