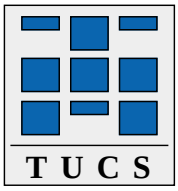
Generic Wrapping

Martin Büchi

Turku Centre for Computer Science
Lemminkäisenkatu 14A, FIN-20520 Turku
Martin.Buechi@abo.fi, http://www.abo.fi/~Martin.Buechi/

Wolfgang Weck

Oberon microsystems Inc.
Technoparkstrasse 1, CH-8005 Zürich
weck@oberon.ch, http://www.abo.fi/~Wolfgang.Weck/



Turku Centre for Computer Science
TUCS Technical Report No 317
April 2000

ISBN 952-12-0569-5
ISSN 1239-1891

Abstract

Component software means reuse and separate marketing of pre-manufactured binary components. This requires components from different vendors to be composed very late, possibly by end users at run time as in compound-document frameworks.

To this aim, we propose generic wrappers, a new language construct for strongly typed class-based languages. With generic wrappers, objects can be aggregated at run time. The aggregate belongs to a subtype of the actual type of the wrapped object. A lower bound for the type of the wrapped object is fixed at compile time. Generic wrappers are type safe and support modular reasoning.

This feature combination is required for true component software but is not achieved by known wrapping and combination techniques, such as the wrapper pattern or mix-ins.

We analyze the design space for generic wrappers, e.g. overriding, forwarding vs. delegation, and snappy binding of the wrapped object. As a proof of concept, we add generic wrappers to Java and report on a mechanized type soundness proof of the latter.

Keywords: Component software, late composition, type systems, language design, generic wrappers, mix-ins, Java

TUCS Research Group
Programming Methodology Research Group

1 Introduction

Component software enables the development of different parts of large software systems by separate teams, the replacement of individual software parts that evolve at different speeds without changing or reanalyzing other parts, and the marketing of independently developed building blocks. Components are binary units of independent production, acquisition, and deployment [53].

Component technology aims for late composition, possibly by the end user. Compound documents, e.g. a Word document with an embedded Excel spreadsheet and a Quicktime movie, as well as Web browser plug-ins and applets are examples of this. Late composition is a major difference between modern components and traditional subroutine libraries, such as Fortran numerical packages, which are statically linked by the developer.

Flexible late composition is one goal, prevention of unsafe compositions, such as adding scroll bars to a file descriptor, leading to 'method not understood' errors and possible system malfunction, is the other. Type systems can help to prevent this kind of run-time errors by prohibiting unsafe compositions. However, static type systems in class-based languages like Java [21], C++ [48], and Eiffel [33] tend to promote inflexible composition mechanisms, such as inheritance, which is fixed at compile time for a whole class.

Untyped prototype-based languages such as Self [54] are more flexible. Here, inheritance relationships can be decided at run time on a per-object base. The price of this flexibility is the lack of certain compile-time and as-early-as-possible run-time error detection: Assignments that may later on cause 'method not understood' errors don't cause any errors at compile time or at the time of their execution. Rather, the errors occur much later when the method is called. Flagging an error at compile time is preferable because it happens in presence of the programmer. Run-time errors, on the other hand, might occur at the clients' sites. Even in this case, trapping as close as possible to the place, where things started to go wrong, greatly facilitates debugging. Furthermore, component-wise (modular) reasoning, another requirement for independently developed components [53], is practically impossible in very flexible prototype-based languages.

In this paper we present an inflexibility problem in class-based languages and propose a new solution that partly borrows from prototype-based languages yet retains the possibility for maximal static and as-early-as-possible run-time error detection and modular reasoning.

Late composition is most pressing for items defined by different components, which may themselves be combined by an independent assembler or even by the user at run-time. Component standards such as Microsoft's COM [45], JavaBeans [50], and CORBA Components [40] are on the binary level. Components can be created in any language for which a mapping to the binary standards exists. However, binary standards are most easily programmed to in languages that support the same composition mechanisms. Furthermore, only direct language-level support can provide the desired machine checkable safety using types. Hence, composition

mechanisms in programming languages are relevant, even though components are binary units.

The mechanism suggested in this paper is partly inspired by COM's aggregation, but it doesn't yet have an exact equivalent in any of the aforementioned binary component standards.

Overview Section 2 illustrates with examples a problem of existing composition mechanisms and defines the requirements for a better solution. In Sect. 3, we show why existing technology does not sufficiently address these requirements. We introduce generic wrappers as a solution to the aforementioned problems in Sect. 4. Next, we discuss the design space for generic wrappers in Sect. 5 and the interplay with other type mechanisms in Sect. 6. As a proof of concept we add generic wrapping to Java in Sect. 7 and report on a mechanized type soundness proof of the extended language in Sect. 8. Section 9 introduces reflective mix-ins as an alternative to generic wrappers. Finally, Sect. 10 points to related work and Sect. 11 draws the conclusions.

2 The Problem

In this section, we describe some applications that cannot be satisfactorily realized with existing composition mechanisms. We also introduce some terminology, and distill a set of requirements.

2.1 Examples

We consider a problem in the realm of compound documents and then show that the same difficulties arise in many other domains.

Embedded views in compound documents for on-screen viewing, such as an Excel spreadsheet in a Word document, may be so large that they require their own scroll bars. Likewise, the user may want to add borders or identification tags to embedded views. It is even possible, that a user wants several such decorators added to the same embedded view.

There may exist different scroll bars from different vendors, which don't know all the other decorator or embedded view vendors. Decorators are typical examples of third-party components that users want to select to meet their specific needs. One user may want proportional scroll bars, another may like blinking borders to draw the boss' attention to the excellent sales figures, and still another may require immutable 128-bit identification tags.

In a compound-document framework similar to Java Swing or Microsoft OLE, let IView be the interface implemented by all classes whose instances can be displayed on screen and inserted into containers. Typical examples of classes implementing IView are TextView, GraphicsView, SpreadsheetView, and ButtonView.

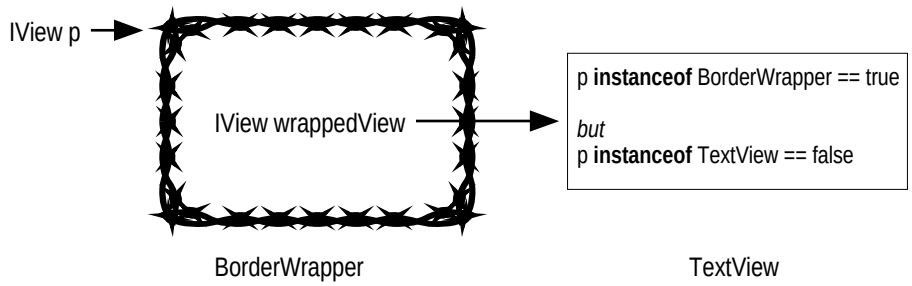


Figure 1: The wrapper is not fully transparent to clients of the embedded view

One way to implement decorators is with wrappers [20, Decorator Pattern]. A border wrapper is itself a view, that is it implements the IView interface. Hence it can itself be inserted into a compound document container. Furthermore the wrapper contains a reference of type IView to a wrapped view, which in a specific instance may be a TextView. The wrapper forwards most requests to the wrapped view, possibly after performing additional operations such as drawing the border.

Unfortunately, this approach has a serious disadvantage. If we wrap a border around a TextView, then the aggregate is only a BorderWrapper, but not a TextView with all of the latter's methods (Fig. 1). Hence, a spell check operation on all embedded text views in a document will not recognize a bordered TextView as containing text, unless it knows how to search inside wrappers from different manufacturers.

A standard interface, like IViewWrapper to be implemented by all view wrappers could ease the problem of searching inside different wrappers:

```
interface IViewWrapper {
  IView getWrappee();
}
```

However, instead of a simple type test, the spell checker would have to loop through all the wrappers:

```
IView q=p;
while(!(q instanceof TextView) && q instanceof IViewWrapper) {
  q=((IViewWrapper)q).getWrappee();
}
if(q instanceof TextView) {...;}
```

This solution is cumbersome for several reasons: First, it requires 5 lines of code instead of a simple type test. Second, it only works if there is a unique standard for wrappers, such as IViewWrapper. Third, it doesn't let the wrapper maintain invariants ranging over both itself and the wrapped object because clients have direct access to the latter.

To work with any type of object, this approach would require a wrapper interface to be defined for any reference type, instance of which might possibly be wrapped. Still the spell checker should be able to locate a wrapped TextView with the above code, whether the wrapper implements IViewWrapper or ITextViewWrapper. Hence, ITextViewWrapper must be a subtype of IViewWrapper, which is only the case in languages that allow covariant specialization of method return types (i.e. of getWrappee) in subclasses. Using a single wrapper interface that defines the return type of getWrappee to be Object would result in a loss of static type information. Parametric polymorphism with covariant subtyping and run-time type information solves this problem, but doesn't address the above three shortcomings.

Support for certain common kinds of wrappers may also be built into the wrapped objects. For example, JComponent, the correspondence to our IView in Java Swing, supports borders as insets. However, identification tags and other kinds of wrapper that were not previewed by the Swing designers are left out.

As a second example, let us consider a forms container that requires all its embedded views to implement the interface IControl. Assume that ButtonView implements IControl and that BorderWrapper doesn't. Hence, a bordered ButtonView cannot be inserted into a forms container: The type system rightfully prevents us from passing a BorderWrapper wrapping a ButtonView as the first parameter to the method insert(IControl c, Point pos). Otherwise a 'method not understood' error could occur when the container tries to call one of the methods declared in IControl. Passing just the wrapped ButtonView as parameter to insert is not a solution, because we would loose the border. The only workaround is to change the type of the first parameter of insert to IView and test that the actual parameter implements IControl or wraps an object that does so. Furthermore, we then either have to store two references per embedded view, one to the outermost wrapper and one to the object implementing IControl, or have the container loop through all the wrappers each time it wants to call a wrapped view's method declared in IControl.

Examples of wrappers in different applications are abundant. Documented cases include window decorators [20], the Microsoft AFC wrapper for AWT components [14], the view wrapper ComponentView for inserting AWT components into Java Swing texts [49], a physical access control system that adds surveillance with wrappers [22], a wrapper that adds additional relations [1], and the stream decorators in the Java library [49]. Many of these applications could be generalized and additional problems could be tackled with wrappers if the problems described above would be solved.

2.2 Terminology

We use the following terminology: A wrapped object is called a *wrappee*. A wrapper and a wrappee together are referred to as an *aggregate*. The declared type of a variable is referred to as *static* (compile-time) *type*. The type of the actually referenced object is called the variable's *dynamic* (actual, run-time) *type*. Likewise, we distinguish between the *static* (declared, compile-time) and the *dynamic* (actual,

run-time) *wrappee type*. For example, for an instance of BorderWrapper, declared to wrap an IView, and actually wrapping a TextView, the static wrappee type is IView and the dynamic wrappee type is TextView.

In discussions, we use the notation C.m to refer to the implementation of instance method m in class C. The subtype relation is taken to be reflexive; e.g., TextView is a subtype of itself.

Except where otherwise stated, the discussion in the first 6 sections applies to most strongly-typed class-based languages such as Java, Eiffel, and C++. For simplicity, we use Java terminology throughout the paper. A Java interface corresponds to a fully abstract class in Eiffel and C++.

2.3 Requirements

From the above examples we can distill a number of requirements for a wrapping mechanism. Numbers in parentheses refer to the summary of requirements in Fig. 2.

The user wants to select which border to wrap around which view. At compile time, the implementor of BorderWrapper doesn't know whether an instance of her class will wrap a TextView, a GraphicsView, or any other view that might even be only implemented in the future. Thus, the actual type and instance of the wrappee must be decidable at run time (1). Furthermore, wrappers must be applicable to any subtype of the static wrappee type (2). In this paper we only consider wrapping of specific instances (selective wrapping), but not adaption of all instances of a given class (global wrapping).

An aggregate of a BorderWrapper wrapping a ButtonView should be insertable into a controls container, even though only the wrappee implements the required interface IControl. Therefore, an aggregate should be an element of the wrapper and the actual wrappee type (3). This also implies that all methods of the wrappee can be called by clients and that they can make these calls directly on a reference to the wrapper.

Upon calling paint on an aggregate of a BorderWrapper and a TextView, the border's paint method should be executed. The latter first draws the border and then calls the paint method of the wrapped view with an adapted graphics context. Thus, wrappers must be able to override methods of the wrappee (4).

If clients can have direct references to the wrappee, they can call overridden methods. For example, a client could call the paint method of the embedded view with the graphics context (dimensions) of the whole aggregate. Thence, a wrapper should be able to control whether clients can directly access the wrappee (5).

A wrapper may depend on the wrappee's state being in a certain relationship to its own, as expressed by an invariant ranging over both state spaces. By overriding methods of the wrappee that could be used to invalidate this invariant and not granting clients direct access to the wrappee, this invariant can be partly protected. However, the actual wrappee type may always provide additional methods that

1. *Run-time applicability.* The actual type and instance of the wrappee must be decidable at run time.
2. *Genericity.* Wrappers must be applicable to any subtype of the static wrappee type.
3. *Transparency.* An aggregate should be an element of the wrapper and the actual wrappee type.
4. *Overriding.* Wrappers must be able to override methods of the wrappee.
5. *Shielding.* A wrapper should be able to control whether clients can directly access the wrappee.
6. *Safety.* The type system should prevent as many run-time errors as possible statically and signal errors as early as possible at run time.
7. *Modular reasoning.* Modular reasoning should be possible in the presence of wrapping.

Figure 2: Requirements for a Wrapping Mechanism

may be used to invalidate the invariant. Requirement 3 states that these methods are accessible to clients.

Early detection of errors and the possibility for modular reasoning have already been identified as general requirements for component-oriented programming. We state them here as explicit requirements (6 and 7) for the purpose of assessing composition mechanisms.

We may want to put both scroll bars and a border around a text view. Hence, multiple wrapping should be supported. We refrain, however, from explicitly listing this as one of our requirements, because it is satisfied by all surveyed mechanisms.

Finally, it is desirable that classes are not required to follow any coding standards for their instances to be wrappable. Otherwise, instances of classes programmed to different standards and of legacy classes are left out. Since certain coding standards can be established, as shown by JavaBeans, and since certain automatic rewriting —even of binary code— is possible, we consider this as a nice-to-have feature, but do not make it a formal requirement.

3 Why Existing Technology Is Insufficient

In this section we show why existing technology fails to address the above requirements.

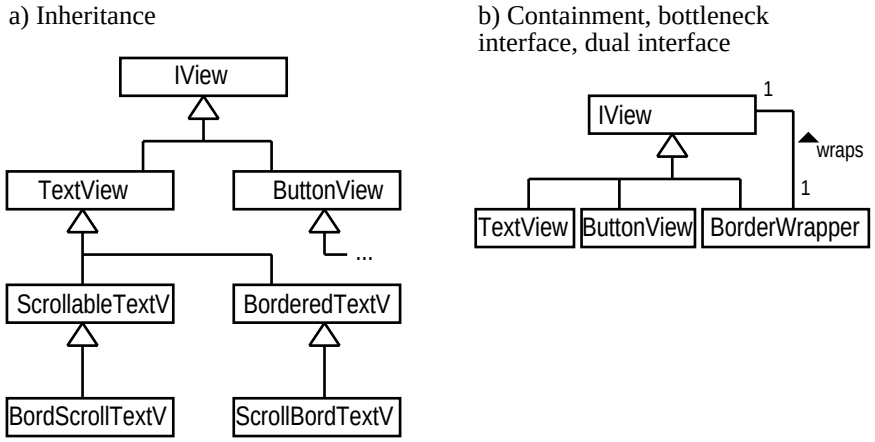


Figure 3: Solution Attempts in Class-Based Languages

3.1 Single-object solutions

Inheritance Feature combination by multiple inheritance produces specialized combination classes, such as BorderedTextView and BorderedGraphicsView. Thus, it combines the functionality of the wrapper and the wrappee into a single object. However, combinations can only be made at compile time by a vendor having access to both the border and the view. Run-time feature composition in interface builders or in compound documents is impossible with inheritance. Hence, this approach fails requirement (1). The modular reasoning requirement (7) is not fully satisfied because of the close coupling between super- and subclass, leading to the semantic fragile base class problem [36]. Furthermore, inheritance suffers from the combinatorial explosion problem, as illustrated by Fig. 3 a.

Mix-ins Parametric/bounded polymorphism, where the type parameter can serve as a supertype of the parameterized type, gives a special form of inheritance. Multiple combinations of wrapper and wrappee types can be created without textual code duplication. With this kind of *mix-ins*[1] [1] we could define the generic border type

```
class ParBorderedView<Wrappee implements IView> extends Wrappee {...}
```

and could derive the class ParBorderedView<TextView> of bordered text views and the class ParBorderedView<ButtonView> of bordered button views. However, also with mix-ins all combinations must be made at compile time. Hence, they fail like normal inheritance the requirement of run-time applicability (1).

[1] Support for mix-ins is rare. Examples include C++ templates, which delay most checking to the derivation of classes, Jigsaw [5], and three extension proposals for Java [1, 18, 3].

3.2 Two-object solutions

Containment The containment approach, also known as the decorator pattern [20], has already been sketched along with the presentation of the example in Sect. 2.1. It is illustrated by Fig. 3 b. The wrapper itself subtypes the static wrappee type and contains a field with a reference to the wrappee. As stated above, this approach fails the transparency requirement (3). Clients can only use the extended functionality of the wrappee by directly accessing it. Thus, the implementor has to choose between the shielding requirement (5) and making the full functionality of the wrappee available to clients.

An additional problem surfaces if the static wrappee type is a class with public fields. In this case, we end up with two unsynchronized copies of these fields in the wrapper and the wrappee.

Specialized wrappers Instead of creating a single BorderWrapper, we could define specialized border wrappers with matching static wrappee types for any kind of view such as TextViewBorder and GraphicsViewBorder. However, this solution attempt fails the run-time requirement (1) for the type: If the border vendor is not aware of SpreadsheetView, there will not be a matching border. Even in a closed system this approach suffers from a combinatorial explosion of classes like the inheritance approach.

Bottleneck interface In the bottleneck approach, the wrapper implements the declared wrappee type and holds a private reference to the wrappee (Fig. 3 b). The difference to the containment approach is that the wrappee only contains a single public method, the message handler, with a parameter containing the instructions what should actually be done. The wrapper also has a message handler method. The latter forwards any message that it doesn't understand itself to the wrappee. This approach does not make good use of the static type system. Fewer errors can be detected at compile time. Run-time type tests cannot be used to determine which messages are understood. Furthermore, callers have to be prepared to handle pseudo method-not-understood return values from the message handler. Thus, errors are not restricted to type casts. In summary, this approach makes the full functionality of the wrappee available to clients, but fails the transparency (3) as well as the safety requirement (6). It requires adherence to special coding standards, but bottleneck interfaces could be generated automatically.

Dual interfaces The containment and the bottleneck interface approach can be combined to get dual interfaces. Wrappees have in addition to their normal public methods a message handler through which all normal methods can be called (Fig. 3 b). Hence, methods of the static wrappee type can be called directly in a type-checked manner and additional functionality of the dynamic wrappee type through the message handler. Dual interfaces fare slightly better than bottleneck interfaces with respect to safety, but otherwise have the same drawbacks.

Technology \ Requirement	Run-time applicability (1)	Genericity (2)	Transparency (3)	Overriding (4)	Shielding (5)	Safety (6)	Modular reasoning (7)
Inheritance		(b)	✔	✔	n/a	✔	(c)
Parameterized mix-ins		(b)	✔	✔	n/a	✔	(c)
Containment	✔	(b)		(d)	(d)	(e)	✔
Specialized wrappers[(a)]	(f)	(b)	✔	✔	✔	✔	✔
Bottleneck interface	✔	✔		✔	✔		
Dual interface	✔	(b)		✔	✔	(e)	
Delegation in prototype-based languages	✔	✔	n/a	✔	✔		

(a) If only used with specific type, otherwise like containment.
(b) Yes, but with exceptions due to signature clashes.
(c) Limited due to tight coupling.
(d) Either full functionality availability or overriding and shielding.
(e) Type safety for static wrappee type.
(f) Type determined at compile time.

Figure 4: Properties of Existing Technologies

Delegation in prototype-based languages Prototype-based languages, such as Self [54], use a parent object to which the receiving object delegates messages that it does not understand itself. A bordered text view could be implemented by a border object with a text view parent. Due to the lack of (static) typing and because of the possibility to reassign the parent object, prototype-based languages fail the requirements of safety (6) and modular reasoning (7) [20].

3.3 Summary

We conclude that none of the existing technologies gives a satisfactory solution to the problem at hand. Figure 4 summarizes the results. Further language specific and binary level solution approaches are described in Sect. 10.

4 Generic Wrappers

To solve the problem stated in Sect. 2, we introduce generic wrappers. Generic wrappers are classes that are declared to wrap instances of a given reference type (class, interface) or of a subtype thereof. Like an extends clause to specify a superclass, we use a wraps clause to state the static wrappee type. This also declares the wrapper class to be a subtype of the static wrappee type. For example, the declaration

```
class LabelWrapper wraps IView {...}
```

states that each instance of the class LabelWrapper wraps an instance of a class that implements IView. The declaration makes class LabelWrapper a subtype of IView. Thus, instances of LabelWrapper can be assigned to variables of type IView

and LabelWrapper has all public members (methods, fields) of IView. Forwarding/delegation of calls to the methods of IView is implicit, that is there is no need to write explicit stubs.

To assure that this subtyping relationship always holds (and thereby that forwarding of calls never fails) must instances of LabelWrapper always wrap an instance of a subtype of IView —already during the execution of constructors. Hence, the wrappee must be passed as a special argument (in our syntax delimited by <>) to class instance creation expressions:

```
TextView t = ...; IView v = new LabelWrapper<t>(...);
```

The compiler checks that the declared type of variable t is a subtype of the static wrappee type. The wrapper class instance creation expression throws an exception if the value of t is null or if t were an expression and its evaluation throws an exception. In both cases, no wrapper object is created and the value of v remains unchanged.

The particularity of generic wrappers is that their instances are not only of the static, but also of the actual wrappee type. For example, a LabelWrapper wrapping a TextView is also of the latter type and not just of type IView. Hence, such an aggregate can be assigned to a variable of type TextView and the latter's methods can be called on it. In the following program fragment, which is based on the definition of LabelWrapper above, the type test returns true and the cast succeeds:

```
IView v = new LabelWrapper<new TextView()>(...);
TextView t2; if(v instanceof TextView) {t2=(TextView)v;}
```

The adjective 'generic' in generic wrapper stands for the reuse of parameterizable abstractions, which we have added to the plain wrapper pattern.

Methods declared in the wrapper override those in the wrappee analogously to overriding in subclasses.

In constructors and instance methods of generic wrapper classes, the keyword wrappee references the wrappee. It can be treated like an implicitly declared and initialized final instance field with some restrictions on use, as discussed below. Hence, wrappers can call overridden methods of the wrappee using the keyword wrappee corresponding to super for overridden methods of superclasses. For example, the paint method of BorderWrapper might look as follows:

```
public void paint(Graphics g) {
  ...; // paint border
  wrappee.paint(g1); // paint wrapped view with adapted graphics context
}
```

A wrapper that is aware of certain subtypes of the static wrappee type, can also use the keyword wrappee in type tests. For example, a wrapper that displays the length of the text in the wrapped view, if the latter is a TextView, might contain the following code fragment:

```
if(wrappee instanceof TextView) {int len=((TextView)wrappee).textLength(); ...;}
```

Preliminary evaluation Although this basic definition still leaves many aspect open, we can evaluate which requirements (Fig. 2) it fulfills independently of how the details are fixed. The actual type and instance of the wrappee can be decided upon at run time. Hence, requirement (1) is satisfied.

Wrappers are applicable to a all of the static wrappee type's subtypes, for which no unsound overriding would occur. An example of the latter might be that a method with signature m() and return type void would be overridden by one with the same signature, but a different return type, as discussed below. Thus, the genericity requirement (2) is mostly fulfilled.

As defined above, instances of generic wrappers are members of the actual wrappee type. Therefore, the transparency requirement (3) is satisfied. Note that none of the surveyed existing mechanisms satisfied both the run-time applicability and the transparency requirement (Fig. 4).

The fulfillment of the shielding (5) and modular reasoning (7) requirements cannot be judged without fixing more details.

The compiler ensures that an aggregate is always of the static wrappee type and, thereby, that all calls to methods of the static wrappee type will succeed. Run-time tests can be used to check whether the aggregate is of a certain type. Only insufficiently guarded casts may fail. Calls to methods of the actual wrappee type always find a matching method. Hence, the type system fulfills the safety requirement (6) by preventing as many run-time errors as possible statically and signaling errors as early as possible at run time.

5 Design Space for Generic Wrappers

The basic definition of generic wrappers in the previous section leaves many aspects open. In this section, we investigate the design space for generic wrappers.

The time of binding has a major influence on the design space of generic wrappers as compared to inheritance. With inheritance, the superclass is bound at compile time. With generic wrappers the actual type and instance of the wrappee first become known at wrap time, that is, run time. Later binding brings flexibility, but means that certain compatibility checks asserting type soundness and, thereby, the success of all method lookups have to be delayed (Fig. 5). A notable feature of generic wrappers is that an existing wrapper object can be wrapped again. Thus, it remains always possible to add new functionality to an aggregate.

Dynamic linking partly blurs this distinction. The name of the superclass is fixed at compile time, but the actual version and, therefore, the members and their semantics are not known until load time. For example in Java, the loading of a class may be delayed until an instance thereof is created. In this case, the compatibility with the used superclass is checked as late as the compatibility between a wrapper and the actual wrappee type. In conclusion, dynamic linking postpones compatibility checking to run time without fully exploiting the flexibility thereof.

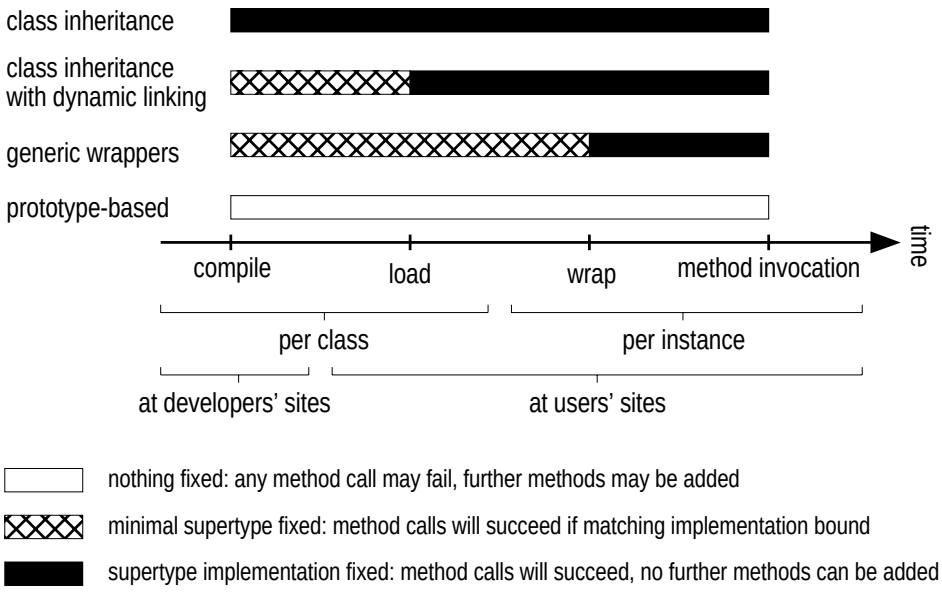


Figure 5: What Is Asserted to Hold from Where on?

5.1 Overriding of instance methods

Overriding of instance methods in subclasses is governed by certain rules to guarantee both type and semantic soundness. The same rules extend to overriding of methods of the wrappee by methods of the wrapper. For example to guarantee type soundness in Java, the overridden method must not be final, the return type of the overriding method must be the same as that of the overridden method, the overriding method must be at least as accessible, the overriding method may not allow additional types of exceptions to be thrown, and an instance method may not override a class method. To also guarantee semantic soundness, the overriding method must be a behavioral refinement of the overridden method [32].

Although the actual type of the wrappee isn't known until wrap time, we can perform certain checks at compile time. We can check that overriding of methods of the static wrappee type by methods of the wrapper respect the above rules. Any violation of the type rules would necessarily also lead to a violation in combination with any actual wrappee type, i.e., a subtype of the static wrappee type.

Because the actual wrappee may have more methods than the static wrappee type, overriding conflicts may nevertheless occur at wrap time, i.e., when the combination of the wrapper and the wrappee first becomes visible. In Fig. 6, three overriding conflicts occur when wrapping an instance of A in an AWrapper. The methods A.m an A.o would be overridden by semantically incompatible ones and AWrapper.n cannot override A.n because they have different return types.

Below we discuss two approaches to this problem. The first checks type soundness at wrap time and throws an exception if wrapping would be type unsound. To

```
interface IA {
  int m(); // return 0 or 1
}

class AWrapper wraps IA {
  public int m() {return 0;};
  public void n() {...};
  public int o() {return 0;};
}
```

```
class A implements IA {
  public int m() {return 1};
  public int n() {...};
  public int o() {return 1};
}

IA a=new A();
AWrapper w=new AWrapper<a>();
```

Figure 6: Overriding Example

decrease the probability of unsound overriding, we suggest a number of coding conventions. The second approach avoids wrap-time type problems by relying on a different form of method lookup and subsumption. We conclude with a short refutation of static approaches.

In this section we assume that there are no final classes and no method header specialization in subtypes (overriding non-final with final methods, overriding with restricted exception throws clauses and higher accessibility, as well as covariant return type and contravariant parameter type specialization) in our language. The interaction of final classes and method header specialization with generic wrappers is discussed in Sect. 6.2.

5.1.1 Wrap-time tests and coding conventions

At wrap time, we can automatically check whether overriding of methods of the actual wrappee by the wrapper is type sound. If this is the case, we can create the wrapper instance. Otherwise, we throw an exception. Wrap-time tests require enough information in the binary code. Java byte code, for example, satisfies this requirement.

Wrap-time exceptions are undesirable, yet they are preferable over unsuccessful method lookup as in prototype based languages like Self. First, if components are combined by an assembler, she can much more easily check all combinations than all method calls on all combinations. Second, if an error occurs, detecting it as early as possible facilitates debugging, as expressed by requirement (6).

To reduce the probability of wrap-time conflicts, we could use laxer rules for wrap-time overriding. For example, Java's binary compatibility prescribes laxer rules for the load-time compatibility checks between a class and its used superclass. In analogy, we could, e.g., allow overriding of a method of the wrappee by a method of the wrapper with an incompatible exception throws clause. Binary compatibility is a last resort for coping with changes to a superclass, fixed at compile time. On the other hand, generic wrappers promote the use of subtypes of the static wrappee type. Furthermore, laxer typing rules threaten semantic soundness, which

must be the ultimate goal. Hence, we believe that the strict rules should be used for generic wrappers at wrap time also.

We suggest to adhere to the following four coding conventions, which can greatly reduce the possibility of both type and semantic conflicts at wrap time:

(a) Classes only define (non private) methods declared in implemented interfaces.

(b) No two interfaces, not related by extension, declare methods with the same signature.

(c) Interfaces have semantic specifications and methods in classes are semantic refinements of their correspondences in the implemented interfaces.

(d) Method calls are only made on variables of interface, but not class types.[2]

We analyze the conventions for method o of Fig. 6. Convention (a) implies that both AWrapper and A implement interfaces declaring a method o. Furthermore, (b) dictates that this must be the same interface, say IO. The idea of class refinement [37], and the related notion of behavioral subtyping [2, 32], is that interfaces have semantic specifications and that methods in subtypes are behavioral refinements of the corresponding methods in their supertype. Assuming that both AWrapper.o and A.o are refinements of IO.o, we can deduce that both 0 and 1 are correct return values. Finally, condition (d) implies that a call x.o() may only be written for x of static type IO. In this case, the value 0 returned by the overriding method AWrapper.m meets our expectations.

If (a) or (b) is not adhered to, then a type conflict may occur as illustrated by method n of Fig. 6. If (c) is not adhered to, it could be that IO.o specifies the return value to be 1, which would not hold in the above case. Finally, if (d) is not respected, we could make a call x.o on a variable of type A. If x contained a reference to an AWrapper wrapping an A, we would get a return value of 0 although we expected 1.

Conventions (a) and (d) could easily be enforced by a programming language. Instead of (b) a language can require qualified notation for member access instead of merging namespaces of interfaces. Convention (c) requires semantic proofs and is, therefore, more difficult to check. These conventions also avoid semantic problems in the overriding in subclasses. Hence, they are implicitly advocated as good style for object-oriented programming [20, 53, 15] and correspond to Microsoft COM's rules/guidelines for the binary level.

In conclusion, wrap-time checking allows us to avoid type unsound overriding. Furthermore, adherence to some also otherwise beneficial coding conventions can greatly reduce the possibility of type or semantic unsound overriding.

[2] Self calls, which are of course also allowed, are discussed in Sect. 5.3.

5.1.2 An alternative form of method lookup

An alternative is to have the wrapper only override methods already present in the static wrappee type. In Fig. 6, this would mean that only A.m would be overridden by AWrapper.m.

Instead of overriding additional methods of the actual wrappee, in the example A.n and A.o, we allow an aggregate to contain multiple methods with the same signature and base the dispatch on run-time context information. In the simplest case, the dispatch is based on the static type of the receiver:

```
AWrapper w=new AWrapper<new A()>();
int i; i=w.o(); // executes AWrapper.o, i=0
A a=(A)w; i=a.o(); // executes A.o, i=1
```

In more general cases, the dispatch is not only based on static, but on run-time context information, i.e., an object's history of subsumptions. To illustrate this, assume that method o is declared in interface IO and that both AWrapper and A implement IO. In the following code fragment, added to the above, the static type of the receiver is in both cases IO, but different implementations are executed:

```
IO x;
x=w; i=x.o(); // executes AWrapper.o, i=0
x=a; i=x.o(); // executes A.o, i=1
```

The problem is that there are two occurrences of IO in the aggregate. Thus, we have to choose one for subsumption.[3] Multiple non-virtual inheritance in C++ has a similar semantics.

In languages that do not support final classes or method header specialization (Sect. 6.2), this form of method lookup avoids wrap-time exceptions. However, to also achieve semantic soundness, we still need to adhere to the above four coding conventions. Otherwise, we could execute a.m() in the fragment above and be surprised that we don't get 1 as result. The soundness problems caused by specialization could only be avoided by fully giving up overriding.

Method lookup and subsumption are more complex with this approach. Furthermore, adding this to a single-inheritance language with 'normal' method lookup and subsumption for inheritance, we end up with two different forms of method lookup and subsumption. The technicalities of this approach for compile-time composition of mix-ins can be found in [18].

5.1.3 Refutation of static approaches

Here we briefly discuss why some approaches that avoid possible wrap-time conflicts at compile time and that are based on normal overriding have serious deficiencies.

[3] In our case, we have already subsumed the aggregate to be of type AWrapper, respectively A. A true choice would be needed in the first line if w were of a subtype of both AWrapper and A, e.g., the compound type [AWrapper, A] [7].

Allowing the wrapper to only override methods of the static wrappee type, but not add additional methods would avoid the problem of unsound overriding of additional methods in the actual wrappee, e.g. A.n. However, not allowing additional methods in the wrapper would be a severe restriction, which would greatly reduce the usefulness of generic wrappers. Furthermore, this approach would fail in languages that support final classes or method header specialization (Sect. 6.2).

Negative type information [44, 10, 23] could express that subtypes of IA must not have a method, like n, that might be overridden in an unsound way. However, this would also mean that an aggregate of an AWrapper and a subtype of IA would not be of a subtype of IA and could, therefore, not be referenced by a variable of type IA. Furthermore, negative type information cannot be expressed in type systems of current languages.

Requiring the exact type of the wrappee to be known at compile time, a third approach, would contradict the requirement of run-time applicability (1).

5.2 Hiding of fields and class methods

In many languages, fields and class methods are hidden rather than overridden in subtypes. Hiding of fields, if permitted, is not problematic because the hiding field may have a different type than the hidden field. The static wrappee type is used to access hidden fields in the actual wrappee. Hiding of class methods is usually governed by similar requirements as overriding of instance methods. Thus, the same two options apply.

5.3 Forwarding vs. delegation

The difference between forwarding (also called redirection and consultation) and delegation is the binding of the self parameter in the wrappee when called through the wrapper. With delegation, the self parameter is bound to the wrapper, with forwarding it is bound to the wrappee. Figure 7 illustrates the difference with a client calling method m of the wrappee on a reference to the wrapper.

Forwarding is a form of automatic message resending; delegation is a form of inheritance with binding of the parent (superclass) at run time, rather than at

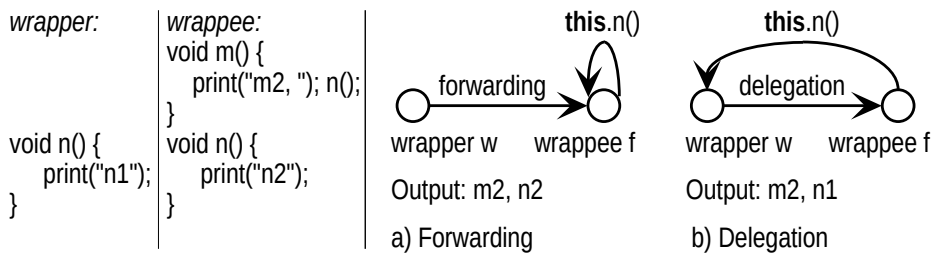


Figure 7: Forwarding vs. Delegation

compile/link time as with ‘normal’ inheritance [31]. Delegation vs. forwarding, the binding time of the parent, and the support for type transparency are almost orthogonal design dimensions.

In all cases, super calls in the wrappee invoke methods of its superclass and not the wrapper’s superclass. During these calls, this is bound to the wrappee with forwarding and to the wrapper with delegation.

The advantage of delegation over forwarding is that the wrapper can better modify and customize the behavior of the wrapped object. The advantage of forwarding is that it eases modular reasoning: As illustrated, delegation can lead to *up-calls* from the wrappee to the wrapper. With forwarding, control stays within the wrappee once a call has been forwarded to it. The wrapper cannot interfere with the flow of control inside the wrappee [53]. Thus, forwarding gives a looser coupling not suffering from the semantic fragile base class problem [36]. This is especially important for component software because the wrapper and the wrappee may be developed independently and composed at run time. Furthermore, forwarding does –unlike delegation– not break encapsulation [46].

5.4 Replacing a wrappee

The wrappee of a given wrapper could be replaced by another object, the type of which is a subtype of the old dynamic wrappee type. It is not sufficient that the new wrappee is a subtype of the static wrappee type: A BorderWrapper wrapping a TextView can be referenced by a variable of static type TextView. Replacing the wrappee by a ButtonView would violate type soundness.

Let ExtTextView be a subclass of TextView. Then, a border wrapper wrapping an ExtTextView should by subsumption be treatable like a border wrapper wrapping a TextView. If we allow a wrappee to be replaced, this is no longer the case. Replacing the ExtTextView in the first aggregate by a TextView is unsound whereas replacing the TextView in the second aggregate by another TextView is sound.

Although cyclic wrapping is type sound in combination with certain features, it is for semantic reasons mostly undesirable. Cyclic wrapping is prevented by the construction process, because the wrappee must be passed as an argument to the wrapper instance creation expression (Sect. 4). If we don’t want cyclic wrapping, we also have to prevent it in the replacement of a wrappee.

For semantic reasons, we think that the wrappee should not be exchangeable. By fixing the wrappee for the lifespan of the wrapper, the system becomes more static and, therefore, simpler to analyze and reason about.

5.5 Direct client references to the wrappee

There are both advantages and disadvantages to allowing clients to hold direct references to the wrappee and being able to invoke the latter’s methods —bypassing a possible overriding by the wrapper. On the positive side, this may give clients the possibility to invoke methods that are ‘accidentally’ overridden. For example, both

BorderWrapper and TextView may define instance methods setColor with the same parameters. Without direct access to the wrappee, clients may not be able to change the text color. With direct access to the wrappee, this is possible. However, only clients that are aware that they reference a wrapped TextView rather than a bare one can do so. With the alternative method lookup (Sect. 5.1.2), TextView.setColor can also be accessed through a cast, unless the method is already declared in the static wrappee type IView.

The disadvantage of direct client references to the wrapper is that it gives an additional way for clients to invalidate invariants ranging over both the wrapper and the wrappee. Furthermore, we end up with different reference values to the same aggregate; thus, loosing the unique identity and the possibility of direct reference comparison.

On a middle ground, we could allow clients to access overridden or hidden members of the wrappee using a special qualified syntax, e.g. w.wrappee.getSize(), but not allow direct references. That is, x=w.wrappee would not be legal. This approach restricts direct client access to few in the source code well visible places and solves the problem of different reference values to an aggregate. To safely access wrap-time overridden members, we would additionally need run-time type tests of the wrappee by clients and casts in the qualified access. For example, to invoke the method setColor, not present in IView, of a wrapped TextView, we would write:

```
if(w.wrappee instanceof TextView) {((TextView)w.wrappee).setColor(c);}
```

If we in principle permit direct client references to the wrappee, we can still let the developer of a wrapper decide for each wrapper class or instance, whether to actually allow such references.

Analogies to overriding in inheritance shows that all of the above options are used in some languages: Java does not allow clients to access overridden methods. In Self, clients can have direct references to parent objects. Finally, overridden methods can be invoked by clients using qualified accessors in C++.

The transparency of generic wrappers reduces the need for direct client references. In the containment approach (Sect. 3.2) all functionality that the dynamic wrappee type provides beyond the static wrappee type can only be made accessible by giving clients direct access to the wrappee. With generic wrappers, on the other hand, the full functionality of the dynamic wrappee type is available through the wrapper —except for accidentally overridden methods.

Whether we allow the wrapper to hand out direct references or not, we have the problems of existing references to the wrappee and of the wrappee handing out self references. Even if we in principle permit direct references, we may want to restrict them to clients that explicitly ask for them and are aware of the dangers.

5.5.1 Redirection of existing references

The problem of existing references vanishes if there aren't any. In analogy to aggregation in Microsoft's COM (Sect. 10), we could require the wrappee to be created along with the wrapper and not allow the wrappee's constructor to pass out self references. The latter condition can, however, only be checked with a semantic proof that can in general not be performed automatically. Furthermore, experience with COM showed that this approach is often too restrictive [42].

The second best case is a single reference to the object to be wrapped. In type systems with aliasing control [25, 12] that can guarantee uniqueness of references we could restrict wrapping to unique references. This single existing reference to the wrappee, which is used as argument in the wrapper construction, could then either be redirected to the wrapper or be set to null. The restriction to unique references may severely limit the applicability of wrappers. Furthermore, aliasing control is not common.

For mainstream languages we see the following options:

1. Keep the references to the wrapped object unchanged. This is only an option if we allow direct references to the wrappee. Unfortunately, clients won't recognize if an object they refer to has been wrapped. Hence, they might unknowingly invoke overridden methods of the wrappee and, thereby, cause the aforementioned semantic problems.

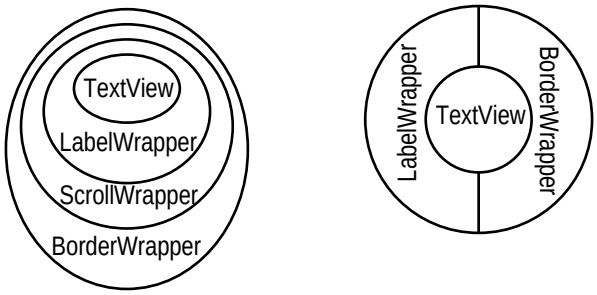
2. Set all existing references to the wrappee to null. Although this approach is type sound, it is clearly unacceptable.

3. Update all existing references to point to the wrapper. Thanks to the transparency of generic wrappers this is sound. Since the type of a reference can only be increased by wrapping, assumptions —gained using run-time type test— that a reference is at least of a certain type are not falsified. On the other hand, negative type assumptions may be invalidated.

5.5.2 Handing out of self references

Wrappees may pass out self references, e.g. for event listener registration. If we don't want direct client references to the wrappee or only allow the wrapper to hand them out, we need to address this issue. The draconian solution is to disallow the use of this in the wrappee except for member access. This is, however, very restrictive and excludes instances of legacy classes not adhering to this restriction from being used als wrappees.

Alternatively, we may define this in the wrappee to reference the wrapper except when used for member access. In combination with forwarding or with delegation and direct client calls of wrappee methods, we get a little semantic curiosity: For variable x of the wrappee's type, this.m() invokes the wrappee's implementation of m() and x=this; x.m() calls the wrapper's overriding implementation, if the latter exists.



a) Conjunctive wrapping b) Disjunctive wrapping

Figure 8: Conjunctive and Disjunctive Wrapping

5.6 Multiple wrapping

There are two forms of multiple wrapping, *conjunctive* and *disjunctive* wrapping (Fig. 8). Conjunctive (also called additive or recursive) wrapping applies multiple wrappers around each other. For example, we might wrap a TextView in a ScrollWrapper and the latter with a BorderWrapper. Cyclic wrapping has been discussed in Sect. 5.4. Infinite wrapping chains are not a problem in practice. They could only occur in infinite executions on computers with infinite amounts of memory.

Disjunctive wrapping presents the same wrappee with different wrappers. It has analogous drawbacks as direct client references to the wrappee, namely the possibility of invalidating invariants ranging over the wrappee and one of its other wrappers. The application of disjunctive wrappers is tricky: Automatic redirection to the outermost wrapper (Sect. 5.5) doesn't work, because there is no single outermost wrapper. Furthermore, disjunctive wrapping is only possible if we allow direct client references to the wrappee, provide a special statement for the simultaneous application of multiple wrappers, or let an existing wrapper wrap its wrappee reference without updating it. With type transparency, disjunctive wrapping can in most cases be replaced by conjunctive wrapping because the full dynamic wrappee type is visible through all wrappers.

If we allow direct client references to the wrappee but not disjunctive wrapping, we have to define what happens if a client wraps an object that is already wrapped. The options are disallowing it and throwing an exception if tried, putting the new wrapper between the wrappee and the old wrapper, and applying the new wrapper around the old wrappee. Thanks to the transparency of generic wrappers, all options are type sound.

5.7 Concealment

In certain cases, a wrapper may want to conceal part of the wrappee from clients. For example, a ConfidentialWrapper and its wrappee should not be serializable for confidentiality reasons. Thus, the wrapper wants to conceal interface Serializable

from clients in case the wrappee implements it. For this case, a conceals clause may be useful in combination with wraps:

class ConfidentialWrapper **wraps** IView **conceals** Serializable {...}

With this definition, no instance of a ConfidentialWrapper aggregate will ever be an element of Serializable. That is, for a variable x referencing such an aggregate, x instanceof Serializable will be false.

Alternatively, a wrapper could be transparent for explicitly listed types only:

class SpecialWrapper **wraps** IView **hoists** IText, IGraphics {...}

When such a SpecialWrapper wraps an instance of a class implementing IText then the functionality declared in IText can be accessed through the wrapper. On the other hand, if the same class also implements another interface, say IContainer, the latter's functionality cannot be accessed through the wrapper and the wrapper cannot be assigned to a variable of static type IContainer. Although transparency is restricted, this approach differs from the containment approach (Sect. 3.2) in that the type of the aggregate depends on the actual type of the wrappee.

Concealment may be practical for special cases, but it causes type soundness problems because the aggregate is not a subtype of the wrappee. Existing references to the wrappee cannot be redirected to the wrapper, if the latter conceals (part of) the static type of the variable containing the reference. Concealment also causes similar problems in combination with solutions 2 and 3 of applying a wrapper to an already wrapped object (Sect. 5.6). Furthermore, with delegation self calls of the wrappee to methods that are concealed by the wrapper fail. For this, a workaround would be to conceal types only from clients, but not from the aggregate itself.

These problems may, but do not necessarily occur in a given system that uses concealment. In analogy to Eiffel allowing subclasses to conceal[4] inherited members, we could allow concealment of types. This would, however, require system validity checks of complete systems.

5.8 Multiple wrappees

So far, we have assumed that a given wrapper instance wraps exactly one object. This could be generalized to a fixed or arbitrary number of objects, thereby providing a single view of a subsystem implemented by multiple objects corresponding to the facade pattern [20]. Similar to multiple code inheritance, this works well unless different wrappees implement methods with the same signature and the wrapper does not override them. In this case, message lookup needs to be redefined. Similar to wrap-time overriding (Sect. 5.1), such conflicts caused by type transparency may not be visible at compile time.

[4]This is called 'hiding' in Eiffel. We don't use this term here to avoid confusion with Java style hiding of class methods and fields (Sect. 5.1).

There are four partly combinable approaches for augmenting the definition of method lookup. The first two are similar to possible solutions for type-sound wrap-time overriding.

1. We can disallow ambiguous aggregates by checking for static conflicts at compile time and throwing an exception at run time when trying to wrap objects that would result in an ambiguity. This approach, however, fails the genericity requirement (2).

2. We can leave the problem unresolved until an ambiguous method is called and only then throw an exception. Self uses this approach in presence of multiple parents. This fails the requirement of as-early-as-possible error detection (6).

3. Message lookup proceeds according to a certain strategy (depth first, breadth first) and a certain order (declaration order, alphabetic order of wrappee type names, etc.). The first matching method is chosen. Any such strategy and order would be rather arbitrary, as illustrated by the criticism of CLOS using syntactic order to choose the correct multi-method [11].

4. The wrapper explicitly defines a lookup strategy and order for conflict resolution. This solution is rather complex and may still not have the desired effect.

6 Interaction With Other Typing Mechanisms

In this section we discuss the interaction of generic wrappers with other common typing mechanisms.

6.1 Subclassing

Here we investigate whether and how generic wrappers can substitute inheritance and how the two may be combined.

6.1.1 Generic wrappers as a substitute for inheritance

If we choose delegation (Sect. 5.3) for generic wrappers, then they can be used to simulate class-based inheritance as follows:

class D **extends** C {…};	**class** D **wraps** C {…};
D d=**new** D();	D d=**new** D<**new** C()>();
Inheritance	Simulation with generic wrappers

The main difference is that at compile time we only know the lower bound of the wrappee type for generic wrappers, whereas with inheritance we know the exact superclass. This can be interpreted as flexibility or as lack of knowledge.

If, on the other hand, we use forwarding instead of delegation for generic wrappers, then we cannot modify the semantics of self calls in methods of the supertype. Thus, such generic wrappers cannot be used to simulate inheritance. Considering the advantages of the looser coupling, generic wrappers might still be used in a language as a replacement for inheritance.

6.1.2 Subclassing of wrapper classes

In most mainstream languages like Java, Eiffel, and C++ subclassing implies subtyping. For this principle to extend to wrappers, a subclass of a wrapper class C has to be declared to wrap the same type X as C or a subtype of X. Covariant specialization of the static wrappee type is possible unless the wrappee can be replaced (Sect. 5.4) using a method like setWrappee(StaticWrappeeType w), where the static wrappee type occurs in a contravariant position.

A seeming alternative to a subclass D of a wrapper class C being itself a wrapper would be that D implements the static wrappee type of C. For example, if C wraps X then D implements X would suffice. However, methods of C may contain accesses to wrappee members using the keyword wrappee. These accesses would be undefined in D because instances of D do not have a wrappee. It would be type sound, but semantically undesirable, to let wrappee be a self reference in such cases. Hence, implementing a superclass' static wrappee type instead of wrapping an object of that type is not an alternative.

Forcing subclasses of wrapper classes to be wrappers themselves implies a restriction on the legal static wrappee types. Let C be a wrapper class with static wrappee type X. Then X must not be equal to C or be a subclass of C. If X is itself a wrapper class, it must not —directly or transitively— be declared to wrap C or a a subclass of C. Only infinite and circular chains, which we both forbid, could be elements of a class C not adhering to these rules. It is, however, for example legal for a BorderWrapper to have the static wrappee type IView (or even some other wrapper type) and have an instance of BorderWrapper wrap another BorderWrapper.

6.2 Method Header Specialization and Final Classes

Some languages allow overriding methods to have more specialized headers. For example, Java allows a non-final method to be overridden by a final one and allows the overriding method to have a more restricted exception throws clause and a higher accessibility. Other languages also allow covariant return type and contravariant parameter type specialization.

This creates problems with overriding by wrappers, even if the overriding is statically visible. Wrapping an instance of B with a BWrapper in Fig. 9, would

```
interface IB {
  void p() throws SomeException;
}

class BWrapper implements IB wraps IB {
  public void p() throws SomeException {...};
}

class B implements IB {
  public final void p() {...};
}

IB b=new B();
BWrapper w=new BWrapper<b>(); // illegal wrapping caught by exception
((B)w).p() // final method would be overridden and exception might be thrown
```

Figure 9: Method Header Specialization Example

override the final method B.p with an empty throws clause by BWrapper.p, which may throw SomeException.

To prevent such unsound overriding, we have to use wrap-time exceptions. Thus, in languages with method header specialization, even the alternative form of method lookup (Sect. 5.1.2) cannot fully avoid the problem of wrap-time exceptions. Method header specialization is the type correspondent of semantic refinement discussed in Sect. 5.1.1.

Final classes pose a similar problem. They should not be subtyped. Thus, it is a compile-time error to declare a wrapper with a static wrappee type that is a final class type. At run time, an exception is thrown if an attempt is made to wrap an instance of a final class.

6.3 Overloading resolution

Many languages support overloading of method names, that is classes containing multiple methods with the same name, but different numbers or types of parameters. For every call, the signature of the method to be invoked is determined at compile time. Compile-time selection of the invoked method's signature means that a better fitting signature of the run-time type, e.g. a subclass, is ignored. The same principle applies to generic wrappers: There is no need for a costly search of a possibly better fitting signature in the actual wrappee during method invocation.

6.4 Parametric types

Parametric types and methods, like C++ templates and generic Eiffel classes, allow compile-time reuse of generic classes and interfaces by providing type parameters and, thereby, creating generically derived classes. Java doesn't support parametric

types. We use here the C++ like syntax with bounds common to most proposals (e.g. [1]) for adding F-bounded polymorphism to Java.

Generic wrappers and parametric types can be combined without problems. Instances of generically derived classes don't distinguish themselves from instances of normal classes. Hence, they can be normally wrapped.

Generic wrappers can be parameterized. As in combination with inheritance, the type parameter might be implicitly limited by soundness constraints for overriding and hiding. Let classes C and D be defined as follows:

```
class C {
  void m(String s) {...}
}

class D<T> wraps C {
  int m(T s) {...}
}
```

Using String for the parameter T, i.e. D<String>, we would get two methods m(String s). Hence, such a derivation has to be forbidden at compile time. An analogous problem occurs in combination of parametric types with inheritance rather than generic wrapping, as illustrated by replacing wraps by extends in the declaration of D above.

By allowing the type parameter in the wraps clause we can make use of possible additional compile time knowledge about the wrappee. Compare the classes E1 and E2, where M stands for some member declarations:

```
class E1 wraps I {M}
class E2<T implements I> wraps T {M}
```

Any legal wrappee of an instance of a derived class of E2 is also a legal wrappee of an instance of E1. However, derived classes of E2, such as E2<C> (assuming that C implements I), can be used to give more static type information. Similar static type information can be provided by a subclass of E1 with covariantly specialized static wrappee type C or with compound types [7], e.g. [E1, C].

Generic wrappers can also be used as bounds in generic classes and as actual parameters in generic derivations.

6.5 Compound types

Compound types [7] let us express directly that the type of a parameter must subtype a set of named reference types, thereby optimally supporting flexible behavioral typing.[5] For example, the compound-typed variable [ILabel, IText] v may reference an instance of a class that implements both ILabel and IText, or the value of v may be null.

[5] Cecil's [11] greatest lower bound types, written ILabel & IText, and Objective-C's [38] multiple protocols, written <ILabel, IText>*, are similar to compound types in Java.

```
signText(Key privateKey, [ILabel, IText] IdText) {...}

class LabelWrapper implements ILabel wraps IText {...}

Text t; ...;
signText(k, ([ILabel, IText]) new LabelWrapper<t>)
```

Figure 10: Summary of example definitions

Compound types let us specify that a parameter must be a text with a label, a text with a border, or even a text with both a label and a border. For example, the method signText(Key privateKey, [ILabel, IText] idText) could set the label to the signature of the text calculated with privateKey (Fig. 10). Let LabelWrapper implement ILabel and wrap IView and let TextView implement IText. An instance of LabelWrapper wrapping a TextView could be passed as second argument to method signText. Without type transparency, this would not be possible.

If the source code of signText were under our control, we could declare the parameter idText to have type ILabel and then in the body of signText get the wrappee and test its type. This approach has, however, several drawbacks: First, the typing of the parameters conveys less precise information. Users don't immediately see what parameters are legal. Second, instances of non-wrapper classes that only implement ILabel are legal parameters. Thus, errors that could be caught at compile time are not. Third, the implementation has to differentiate between parameters that implement the two interfaces in one or two objects. In conclusion, compound types allow for more precise typing, but without type transparency certain aggregates would not be legal parameters.

Constituent types of a compound type being implemented by different objects may, however, lead to undesirable semantic effects. Assume, for example, that TextView also implements ILabel and while the text is modified constantly updates the signature that it itself stores. In this case, modifying the text contents of a TextView wrapped by a LabelWrapper and then reading the label, may unexpectedly returns a wrong signature, namely that stored in the wrapper and not that of the wrappee. The same problem exists, of course, in the decorator pattern (Sect. 3.2).

7 Generic Wrappers in Java

As a proof of concept, we add generic wrapping to Java. We present generic wrappers as a strict extension, that is existing Java programs need not be changed and instances of existing classes can be wrapped.

We select a consistent set of features from the aforementioned design choices and give a definition of generic wrappers in Java. We base our choices on the motivating examples and the above discussions, without repeating the latter. Next, we discuss selected integration issues with the Java library. Finally, we show how the defined mechanism solves the motivating problem.

In the next section, we report on a mechanized type soundness proof for the presented solution. A discussion of efficient implementation strategies is beyond the scope of this paper.

7.1 Feature selection and language integration

Both compile-time and wrap-time overriding and hiding are governed by the same rules as (compile-time) overriding and hiding in subclasses. Furthermore, we don't allow instances of final classes to be wrapped. Violations of these rules by the wrapper/static wrappee pair are flagged at compile time; violations by the wrapper/actual wrappee pair cause exceptions at the time of wrapping.

To get loose coupling between the wrapper and the wrappee and to facilitate semantic reasoning we chose forwarding over delegation and fix the wrappee for the lifespan of the wrapper. All existing references to the wrappee are redirected to the wrapper upon wrapping. We define this in the wrappee to refer to the wrapper except when used for member access. For example, this.x is the field x of the wrappee, but s.register(this) passes a reference to the wrapper as parameter. This approach guarantees a unique identity of the aggregate from the clients' point of view.

In a tribute to flexibility, we allow clients to explicitly attain direct references to the wrappee. Still, we hope this feature proves to be superfluous. The implementor of the wrapper class determines whether clients can get direct references to the wrappee by putting an access modifier (private, protected, public) between the keyword wraps and the static wrappee type, e.g:

```
class LabelWrapper3 wraps public IView {...}
```

The access modifier of the wrappee in a subclass must provide at least as much access as that in the superclass.

The keyword wrappee can be treated like the name of a final instance field of the wrapper class with the used modifier, e.g public in the above example. For example, let x be a variable of type LabelWrapper3. Then clients can access the wrappee as x.wrappee. To navigate back from a wrappee to its outermost wrapper, the method:

```
public final Object getWrapper() {return this;}
```

is added to the class Object. With the above definitions, this method returns a reference to the wrapper if the receiver object is wrapped and otherwise to the receiver itself.

We allow only conjunctive, but not disjunctive wrapping. Wrapping an already wrapped object corresponds to wrapping its outermost wrapper. Because it is not sound in combination with the above features, we don't allow concealment. Every wrapper has exactly one wrappee. (The wrappee may of course itself be a wrapper.)

Although believed not to cause any problems, we do not allow array objects to be wrapped, as would be possible for wrappers, the static wrappee type of which is an array type, Object, Cloneable, or Serializable. The latter are the only interfaces implemented by arrays.

Grammar The grammar for class declarations and class instance creation expressions is augmented as follows [21]:

ClassDeclaration:	Modifiers$_{opt}$ **class** Identifier Super$_{opt}$ Interfaces$_{opt}$ Wrapper$_{opt}$ ClassBody
Wrapper:	**wraps** AccessModifier$_{opt}$ ReferenceType
AccessModifier:	one of **public protected private**
ClassInstanceCreationExpression:	**new** ClassType Wrappee$_{opt}$ (ArgumentList$_{opt}$)
Wrappee:	< Expression >

Additionally, wrappee is added as an alternative to the PrimaryNoNewArray production.

Synchronized methods To simplify synchronization between threads, acquiring a lock on a wrapper instance also locks the wrappee, possibly recursively in case of conjunctive wrapping.

7.2 Library integration

The library being an integral part of Java —the description of three packages is even part of the Java language specification— we discuss how selected features interplay with generic wrappers. Generic wrappers integrate in a straightforward way with most libraries, often providing new possibilities.

Serialization For instances of a Java class to be serializable, the class must implement the empty interface Serializable. The serialization of wrappers is problematic in case the wrapper implements Serializable, but the wrappee doesn't. Type soundness forbids us to not externalize the wrappee and set the wrappee reference to null upon deserialization. We see the following options:

1. Disallow serializable wrappers to wrap instances of non-serializable classes.
   (a) Using compound types (Sect. 6.5), this can be done statically without otherwise restricting applicability: A wrapper class that implements Serializable must require the wrappee to do the same, e.g., a label wrapper implementing Serializable would have to be declared as LabelWrapper wraps [IView, Serializable].
   (b) Expressing this requirement without compound types adds unnecessary restrictions, if the desired wrappee type, e.g. IView, is not a subtype of Serializable. In this case we have to declare a subinterface of IView

and Serializable. However, due to by-name equivalence of types this unnecessarily excludes classes implementing the two directly [7].

(c) Instead of static checks we could resort to throwing an exception when trying to wrap an instance of a non-serializable class by a serializable wrapper as we do for unsound overriding (Sect. 5.1).

2. Treat the wrappee like an object referenced by a field of the wrapper and throw a NotSerializableException when trying to serialize the aggregate. This ruins the clients' perception of the aggregate being a single object.

3. If the wrappee has a no-argument constructor, only serialize the wrapper and create a new wrappee upon deserialization. Otherwise, use one of the above options. This is analogous to a subclass of a non-serializable superclass.

4. Ignore security and other concerns of the implementor of the wrappee and serialize the latter nonetheless.

With compound types, we choose the first option because it solves the problem at compile time without introducing any unnecessary restrictions. Otherwise we'd use the second option.

A dual problem occurs if the wrappee implements Serializable, but the wrapper doesn't. With concealment, we could conceal the interface from clients. Without, we get almost dual options. However, due to the lack of static negative type information there is no correspondence options 1 (a) and 1 (b), except the very restrictive requirement that all wrapper classes must implement Serializable. We, therefore, choose the second option of throwing a NotSerializableException when trying to serialize a wrapper that is not serializable.

Cloning The general contract of clone is that it creates and returns a copy of the receiver object [21]. Because we don't allow disjunctive wrapping, clone of a wrapper has to either create a deep copy or throw a CloneNotSupportedException.

The method clone is defined in Object, classes implement the empty interface Cloneable to indicate that they actually support cloning. If a wrapper implements the interface, but the dynamic wrappee type doesn't, we have a similar problem as for serialization. Analogously we have the options of disallowing a wrapper that implements Cloneable to wrap an instance of a class that doesn't, throw a CloneNotSupportedException, or create a new wrappee with the no-argument constructor if present and accessible. We opt again for the first choice.

The dual problem of the wrappee, but not the wrapper, implementing Cloneable is again solved with the second option. The implementation of clone, that the wrapper inherits from Object and which overrides the implementation in the wrappee, throws a CloneNotSupportedException.

7.3 Assessment

Our mechanism fulfills all requirements (Fig. 2) except for genericity (2). The latter fails in cases where overriding or hiding would not be sound. We consider this acceptable because exceptions are already thrown at the time of wrapping — and not at the time of member access— and because creation of new instances can also fail for other reasons with an exception in existing Java.

Clearly, the motivating problems (Sect. 2.1) can be solved with the presented generic wrappers for Java: Let BorderWrapper be declared as follows:

class BorderWrapper **wraps** IView {...;}

If such a border wraps a TextView, the aggregate is of type TextView and is, therefore, recognized as such by the spell check procedure of all embedded views. Likewise, if such a border wraps a ButtonView, the aggregate is of type IControl implemented by ButtonView. Hence, it can be inserted into a forms container. The developers of TextView, ButtonView, the spell check operation, and the forms container don't have to do any special coding for this to work.

8 Type Soundness

In this section, we report on a mechanically verified formal proof of type soundness of Java extended with generic wrappers. Type soundness intuitively means that all values produced during any program execution respect their static types. An immediate corollary of type soundness is that method calls always execute a suitable method, that is, there are no 'method not understood' errors at run time. Type soundness is not a trivial property, especially for polymorphic languages [6, 9]. It came to prominence with the discovery of its failure in Eiffel [13, 34]. Static typing loses much of its raison d'être if type soundness does not hold.

Our proof of type soundness for generic wrappers is based on the work of von Oheimb and Nipkow [41], a much extended version of [39]. They have formalized a large subset of Java and mechanically proved type soundness with the theorem prover Isabelle/HOL [43].

For this paper, we have added generic wrappers to this formalization. For simplicity, we have extended the formalization of the existing Java type system, rather than our previous extension with compound types [7]. Finally, we adapted the proofs and ran them through Isabelle/HOL.[6]

8.1 Definitions

Here, we present the widening and casting relations, which are interesting in their own rights. Since all type judgments involving arrays are unchanged, they are

[6] At http://www.abo.fi/~mbuechi/publications/GenericWrapping.html the Isabelle theories are available.

omitted in this presentation. A full report of all the mechanical details is beyond the scope of this paper.

The Java language specification introduces identity and irreflexive widening conversions separately. The Java language specification [21] uses the term 'widening' for its form of subtyping. Since identity conversions are possible in all conversion contexts permitting widening, the two are merged in the formalization. The expression $\Gamma \vdash S \preceq T$ says that in program environment Γ objects of type S can be transformed to type T by identity or widening conversion. In particular, expressions of type S can be assigned to variables of type T and expressions of type S can be passed for formal parameters of type T.

We use the following naming conventions:

C, D	classes	A	list of classes
I, J	interfaces	S, T	arbitrary types
R	reference type	Γ	program, environment

The judgment $\Gamma \vdash C \prec_{\mathsf{C}} D$ expresses that class C is a subclass of class D, $\Gamma \vdash C \rightsquigarrow I$ that class C implements interface I, and $\Gamma \vdash I \prec_{\mathsf{I}} J$ that I is a subinterface of J (Fig. 11). Furthermore, $\mathsf{is\_type}\ \Gamma\ T$ expresses that T is a legal type in Γ, $\mathsf{RefT}\ R$ denotes reference type R, and NT stands for the null type.

$\mathsf{Class}\ C$ stands for the class type C and $\mathsf{Iface}\ I$ for the interface type I. Furthermore, the discriminators $\mathsf{is\_class}\ \Gamma\ C$ and $\mathsf{is\_iface}\ \Gamma\ I$ are used.

In our formalization we now have two kinds of classes: normal (non-wrapper) classes and wrapper classes. The discriminator $\mathsf{is\_wrapper}\ \Gamma\ C$ is true if C is a wrapper class and false otherwise. $\mathsf{WrappeeOf}\ \Gamma\ C$ denotes the static wrappee type of class C in program environment Γ.

At run time, instances of wrapper classes are of aggregate types. Aggregate types are finite lists of at least two class types. An instance of the wrapper class C wrapping an instance of the wrapper class D that itself wraps an instance of the (non-wrapper) class E belongs to type $\mathsf{Aggregate}\ [C, D, E]$.

The discriminator $\mathsf{is\_aggregate}\ \Gamma\ A$ is true if A is a list of class names, all but the last element of A denote wrapper classes in Γ, the last element denotes a non-wrapper class, for each $i \in 0..\mathsf{length}\ A - 2$ there exists a $j_i > i$ such that the j_ith element extends, implements, or is equal to the static wrappee type of the ith element, and there are no clashes between method signatures of the elements of A.

$\Gamma \vdash S \preceq T$	S widens to ('is subtype of') T in Γ
$\Gamma \vdash C \prec_{\mathsf{C}} D$	C is a subclass of D in Γ
$\Gamma \vdash C \rightsquigarrow I$	C implements I in Γ
$\Gamma \vdash I \prec_{\mathsf{I}} J$	I is a subinterface of J in Γ
$\Gamma \vdash S \preceq_{?} T$	cast from S to T permissible at compile time in Γ

Figure 11: Summary of notation

Since there are no variables of aggregate type and because we do not allow the dynamic reassignment of wrappees, we only need widening rules with aggregates on the left-hand side of the conclusion judgment.

The following six typing judgments apply unchanged also to wrapper classes:

$$\frac{\mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash T \preceq T} \qquad \frac{\mathsf{is\_type}\ \Gamma\ (\mathsf{RefT}\ R)}{\Gamma \vdash \mathsf{NT} \preceq \mathsf{RefT}\ R}$$

$$\frac{\Gamma \vdash I \prec_{\mathsf{i}} J}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Iface}\ J} \qquad \frac{\mathsf{is\_iface}\ \Gamma\ I; \mathsf{is\_class}\ \Gamma\ \mathsf{Object}}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Class}\ \mathsf{Object}}$$

$$\frac{\Gamma \vdash C \prec_{\mathsf{C}} D}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D} \qquad \frac{\Gamma \vdash C \rightsquigarrow J}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Iface}\ J}$$

The following widening rules involving wrapper classes are used at compile time:

$$\frac{\mathsf{is\_wrapper}\ \Gamma\ C;\ \Gamma \vdash \mathsf{WrappeeOf}\ \Gamma\ C \preceq \mathsf{Class}\ D}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D}$$

$$\frac{\mathsf{is\_wrapper}\ \Gamma\ C;\ \Gamma \vdash \mathsf{WrappeeOf}\ \Gamma\ C \preceq \mathsf{Iface}\ J}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Iface}\ J}$$

The following widening rules involving aggregates are used at run time (set converts a list into a set):

$$\frac{\mathsf{is\_aggregate}\ \Gamma\ A;\ \exists C \in \mathsf{set}\ A.\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D}{\Gamma \vdash \mathsf{Aggregate}\ A \preceq \mathsf{Class}\ D}$$

$$\frac{\mathsf{is\_aggregate}\ \Gamma\ A;\ \exists C \in \mathsf{set}\ A.\Gamma \vdash C \rightsquigarrow J}{\Gamma \vdash \mathsf{Aggregate}\ A \preceq \mathsf{Iface}\ J}$$

The casting relation $\Gamma \vdash S \preceq_? T$ states, that a cast from type S to type T is permissible at compile time, that is, the type cast '(T)e', where e is of type S, might succeed at run time. This is interesting because if it can be proven to always fail, the compiler can already flag an error.

If $\Gamma \vdash S \preceq T$ holds, the cast can be proven to always succeed. Otherwise, a run-time validity test must be performed to check whether $\Gamma \vdash R \preceq T$ holds for the run-time type R of the cast operand. The following general casting conversions are applicable to wrapper classes as well:

$$\frac{\Gamma \vdash S \preceq T}{\Gamma \vdash S \preceq_? T} \qquad \frac{\mathsf{is\_class}\ \Gamma\ C;\ \mathsf{is\_iface}\ \Gamma\ J}{\Gamma \vdash \mathsf{Class}\ C \preceq_? \mathsf{Iface}\ J} \qquad \frac{\mathsf{is\_iface}\ \Gamma\ I;\ \mathsf{is\_class}\ \Gamma\ D}{\Gamma \vdash \mathsf{Iface}\ I \preceq_? \mathsf{Class}\ D}$$

The following two casting rules have weaker conditions in the presence of generic wrappers:

$$\frac{\mathsf{is\_class}\ \Gamma\ C;\ \mathsf{is\_class}\ \Gamma\ D}{\Gamma \vdash \mathsf{Class}\ C \preceq_? \mathsf{Class}\ D} \qquad \frac{\mathsf{is\_iface}\ \Gamma\ I;\ \mathsf{is\_iface}\ \Gamma\ J}{\Gamma \vdash \mathsf{Iface}\ I \preceq_? \mathsf{Iface}\ J}$$

8.2 Theorems and conclusions

With the above definitions we proved that evaluation and execution are type sound and that method lookup always succeeds. These theorems on the extended type system correspond to the ones proved by von Oheimb and Nipkow for Java without generic wrappers. The first two theorems are syntactically equivalent to the ones of von Oheimb and Nipkow. Semantically they are, however, different because the types include generic wrappers. The method lookup theorem is both syntactically and semantically different.

The currently by von Oheimb and Nipkow formalized subset of Java, on which we build, still does not capture all features. Of them final classes, modifiers (currently only static), interface fields, and methods of the class Object would be relevant for generic wrappers. In particular, final classes would allow us to slightly strengthen some of the premises in the above casting rules.

The main advantages of a mechanized over a paper-and-pencil proof are additional confidence and better support for extensions. We would like to stress the second aspect. Not only did the formalization result in a soundness proof, but the proof tool also reminded us of what all needed to be defined about generic wrappers before the desired properties could be established. Most proof scripts worked without modifications. The fact that all theorems were reproved mechanically for the extended language definition conveys more confidence than the typical adaptation of a paper-and-pencil proof with 'this-should-still-hold' handwaving.

9 Reflective Mix-Ins

Mix-ins with capabilities to create new derived classes at run time provide a single-object alternative to generic wrappers. To our knowledge, gbeta (Sect. 10) is the only typed language that supports the derivation of mix-ins from generic classes at run time. However, since gbeta differs greatly from most other object-oriented languages such as Java, Eiffel, and C++, a transfer of this mechanism to other languages is not straightforward. Hence, this section should be understood as an alternative proposal, not as a comparison with existing languages.

9.1 The proposed mechanism

To illustrate the mechanism, consider again the parameterized class ParBordered-View:

class ParBorderedView<Wrappee **implements** IView> **extends** Wrappee {...}

Usually, derivations such as ParBorderedView<TextView> can only be made at compile time. Hence, we concluded in Sect. 3.1 that mix-ins do not satisfy the requirement of run-time applicability (1) as, for instance, required to implement compound documents.

We could, however, allow derivations to be made at run-time. We outline such a proposal for Java based on reflection. In Java the static method forName of Class gets the class object of a class. Assume that this also works for parameterized classes. Thus a reference to the class object of ParBorderedView can be assigned to the variable pc as follows:

```
Class pc=Class.forName(”ParBorderedView”);
```

From this, we can get the class object of a derived class with the postulated method derivation. The type argument of the latter is a class object.

```
Class dc=pc.derivation(Class.forName(”TextView”));
```

If needed, this creates a new derived class. Actual run-time applicability comes from the fact that we can replace the string constant ”TextView” by a variable. Finally, we can create instances of the derived class using the reflection method newInstance:

```
Object o=dc.newInstance();
```

As shown, this can be used to create arbitrary bordered views at run time. Let us assess this solution with respect to the requirements (Fig. 2) and in comparison with generic wrappers.

9.2 Comparison with generic wrappers

Reflective mix-ins have roughly the same properties as generic wrappers that require the wrappee to be created together with the wrapper. Consistency with mix-ins derived at compile time further restricts the design space of reflective mix-ins as compared to generic wrappers.

9.2.1 Combination of wrapper and wrappee into a single object

Reflective mix-ins combine the wrapper and the wrappee into a single object. This has several consequences. First, reflective mix-ins cannot be used to wrap existing objects.

Second, the replacement of a wrappee (Sect. 5.4) and direct client references to the wrappee (Sect. 5.5) are not applicable. On the negative side, the latter implies that accidentally overridden methods cannot be called by clients. On the positive side, the problems of redirecting existing references and of handing out self references don’t exist. However, these problems don’t occur with generic wrappers either, if we force the wrappee to be created along with the wrapper.

Third, the combination of the wrapper and the wrappee into a single object means that only conjunctive, but not disjunctive wrapping is possible (Sect. 5.6).

9.2.2 Various differences and similarities

The combination of the super- and subclass becomes first visible when the corresponding derived class object is generated with the method derivation. Hence, possibly unsound combinations due to overriding and hiding conflicts can only be caught at run time by exceptions as for generic wrappers (Sect. 5.1).

Consistency with mix-ins derived at compile time dictates that we use delegation rather than forwarding. Thus, reflective mix-ins suffer from the semantic fragile base problem. The latter is aggravated by the fact that the base class is not statically known and the combination cannot be analyzed at compile time.

The static type safety of reflective mix-ins and generic wrappers is roughly equivalent. However, the return type of newInstance and all other reflective methods for creating instances from a class object being Object, we always need an initial cast with reflective mix-ins. Furthermore, the parameters of constructors cannot be statically type checked.

An advantage of reflective mix-ins is that they allow the type parameter to be used in other places than just in the extends clause. The usage must, however, be restricted to covariant or even private occurrences to maintain subtyping.

9.2.3 Overloading resolution

Reflective mix-ins cause two kinds of overloading resolution problems that do not occur with generic wrappers. These problems arise from the following two design principles: A generically derived class that is derived at compile time has the same semantics as a plain class with the same members (copy semantics). The semantics of a derived class is independent of the time of derivation (compile or run time). Thus, the copy principles also applies to classes that are derived at run time using reflection. Based on this, we can illustrate the two problems.

```
class C {}
class D extends C {
  void m(Integer x) {...}
}
class X {
  static Object n(Object x) {...}
  static int n(D x) {...}
}
class W<A extends C> extends A {
  void m(String x) {...}
  void o() {
    Object y=X.n(this); // different resolution for W<D> causes type error
    m(null); // resolution ambiguous for W<D>
  }
}
```

Figure 12: Overloading Resolution Problems of Reflective Mix-Ins

First, the most specific method may depend upon the derivation parameter. In Fig. 12, the most specific method for the call X.n(this) is the one with return type Object in the derived class W<C>. However, in the derived class W<D> the most specific method is the one with return type int. In this case the assignment to y is ill-typed.

Second, in languages, such as Java, without a total order between methods for overloading resolution, calls may be ambiguous in certain derived classes. The call m(null) is unambiguous in W<C>, but it is ambiguous in the derivation W<D>.

Like overriding conflicts, changes in overloading resolution and overloading ambiguities can only be caught at the time of derivation by raising an exception. These problems do not exist for generic wrappers (Sect. 6.3). Because combinations with actual wrappee types are only made at run time, there is no need to follow the copy semantics by analogy to a static case. Thus, overloading can be resolved based on the static wrappee type. Hence, generic wrappers fare slightly better than reflective mix-ins with respect to the genericity requirement (2). A more detailed discussion of overloading resolution for static mix-ins can be found in [3].

10 Related Work

Section 3 already provides an overview of some related mechanisms. With the exception of delegation, where a final comparison with our mechanism is deemed interesting, these technologies are not discussed again here.

10.1 Language mechanisms

Delegation in prototype-based languages What do we gain with generic wrappers over delegation in prototype-based languages?

First, the static wrappee type and calls to it can be statically type checked. Some prototype-based languages, such as Cecil [11], also have (optional) static type systems. However, these languages require the exact type or even the concrete instance of the parent object to be known at compile time. The same approach is taken by prototype-based object calculi, e.g. [17]. Thus, they fail the requirement of run-time applicability (1).

Second, with generic wrappers the dynamic wrappee type can be checked with run-time type tests.

Third, type casts are the only points of failure; method lookup always succeeds. This greatly simplifies debugging by indicating errors closer to where they occur.

Fourth, generic wrappers are targeted at mainstream class-based languages.

For our exemplary generic wrappers in Java, we have chosen a set of distinguishing features that facilitate modular reasoning. First we use forwarding rather than delegation. Second the wrappee is assigned snappily differentiating it from reassignable parent fields. Third, we disallow disjunctive wrapping. The latter is no

problem because we get sharing of behavior from classes whereas prototype-based languages have to use shared parents for this.

Lava Kniesel [30] has implemented an extension of Java with wrappers. The main difference to our generic wrappers is that in his proposal the aggregate is not a subtype of the actual, but only of the static wrappee type. Thus his proposal fails the transparency requirement (3) and is more limited in its applicability. Lava's wrappers are a form of the decorator pattern (Sect. 3.2) with automatically generated forwarding stubs and multiple wrappees combined with delegation. Wrappees can be reassigned, thereby, complicating semantic reasoning. The proposal is not type sound because the wrappees are assigned within the constructor. Independent extensibility, the focus of our proposal, is not well supported.

Delegation for software and subject composition Harrison et al. [24] discuss options for different bindings of this in the decorator and facade patterns. They show how to implement delegation using either stored or passed pointers in class-based languages. Furthermore, they propose a declarative approach, to be used by component assemblers, permitting the binding of this to be customized on a per-method base. Their solution does not address the shortcomings of the decorator pattern with respect to our requirements. Namely, it does not provide for transparency (3).

gbeta gbeta [16], a generalized version of BETA, supports two forms of dynamic (parent fixed at run time) inheritance through multiple inheritance. Dynamic object specialization is a dynamic modification of the structure of an existing object, preserving object identity. For example, the statement somePtn##->anObject## enhances the structure of anObject with the pattern somePtn. Furthermore, gbeta allows a form of reflective mix-ins through non-constant virtual types as superpatterns.

Because gbeta uses submethoding with INNER rather than overriding, it is not obvious how the mechanisms of gbeta could be transferred to more 'standard' object-oriented languages.

Dynamic mix-ins Steyaert et al. [47] propose dynamic inheritance through mix-ins. The catch is that each object must contain a specification of all its potential enhancements. This renders their proposal inapplicable for mutually unaware component vendors. Mezini [35] also presents a sophisticated, but complex approach to object evolution without name collisions. However, her work is untyped.

Cecil In addition to the combination of prototypes with an (optional) static type system, Cecil [11] has two more features worth a comparison: predicate objects and multi-methods. Predicate objects are Cecil's more restricted alternative to dynamic inheritance. An object o that inherits from all parents of a predicate object

p automatically also inherits from p if the state of o satisfies the predicate of p. Predicate objects permit important states of objects to be explicitly identified and named. However, with respect to the problem at hand, they are mere syntactic sugar for if or case statements in the methods of the parent objects.

Multi-methods are, ignoring modularization, just elegant syntactic sugar for an explicit coding of a Cartesian product [52]. Since multi-methods can —with certain restrictions to guarantee a best fit [11]— be defined outside the classes of their receivers, they can be used to modify a component without changing the latter's source code [26]. However, they do not address the problems of independent extensibility and run-time applicability. Furthermore, they cannot be used to selectively change the behavior of certain instances only.

Lag*oo*na Lag*oo*na [19] is a single dispatch language that separates messages from reference types. Any message (without a return type) can be sent to any object. For messages without return types, object types can provide a default method with programmable forwarding. Thus, wrappers could simply forward messages that they don't understand to their wrappees. However, only additional methods with return type void can be called. The wrapper is not a subtype of the actual wrappee and type test cannot be used directly to test whether a message will be understood. Hence, Lag*oo*na does not satisfy the transparency requirement (3). With respect to the problem at hand, forwarding in Lag*oo*na is just a syntactically sugared version of bottleneck interfaces.

Fewer errors are caught by the type system because any message can be sent to any object and semantic reasoning is difficult due to the programmable resending.

Objective C Categories in Objective C [38] allow classes to be extended with a new set of methods/protocols independently of the original class definition. This compile-time mechanism corresponds to creating a subclass and globally replacing all occurrences of the superclass by the subclass. Categories modify whole classes, rather than individual objects. Categories do not fulfill the requirements of run-time applicability (1) and genericity (2).

Binary Component Adaption (BCA) BCA [28] provides for similar adaption of Java binaries as categories for Objective-C binaries. Thus, BCA does not solve the problem at hand either.

Aspect-oriented programming Aspects [29] are a new category of programming construct that 'cross-cut' the modularity of traditional programming constructs. So an aspect can localize, in one place, code that deeply affects the implementation of multiple classes or methods. Aspects modify classes at compile time. Hence, they do not address the problems of run-time composition of objects created by different components from different vendors.

Mix-in calculus Bono et al. have developed a formal calculus of classes and mix-ins [4]. Method declarations in mix-ins are explicitly marked as overriding an existing method or introducing a new method. The lower type bound (static wrappee type) is computed from the signature of a mix-in. Redefined methods give positive type information and new methods negative type information. Subtyping is determined by the types' structures. Negative type information is used to avoid mix-in-application-time exceptions.

We believe that name equivalence for types in combination with our coding conventions (Sect. 5.1.1) is better suited to avoid accidental overriding. First, with structural subtyping a method marked as redefining may override an unrelated method that happens to have the same signature. Our solution avoids this. Second, a method m marked as new cannot override a method m from the actual base class even if the two were meant to correspond (as in our system expressed by the fact that the wrapper and the wrappee implement an interface IM declaring m). In our approach, overriding is possible in this case.

The addition of new/redefined method attributes to our generic wrappers would not be very useful. Positive type information can be expressed by the explicitly named static wrappee type. Declaring a method m in the wrapper to be new if the static wrappee type contains a method with the same signature is pointless because it leads to a compile-time error. This leaves us with the possibility to mark a method n as new if the static wrappee type does not contain a method with this signature. For this to be useful, the type system would have to support negative type information. As discussed in Sect. 5.1.3, this is rare and causes other problems.

10.2 Binary component standards

As stated in Sect. 1, wrapping of objects created by different components requires binary standards. Thus, we survey below the most common component standards.

However, even with wrapping on the binary level, direct language-level support has many advantages. First, it makes it simpler and less error-prone to write components for binary wrapping mechanisms, because component instances can be referenced by normal, tightly typed variables and method calls can be type checked. Second, the full power of type systems for early error detection can only be used with programming language support.

Microsoft COM COM is a language-independent binary component standard. It provides two forms of object composition for reuse: containment and aggregation [45]. With containment, the wrapper (outer) holds a reference to the wrappee (inner) and must provide explicit forwarding stubs. Thus, COM containment shares most properties with its language-level sibling (Sect. 3.2).

A COM object implements a set of interfaces. Clients have only references to these interfaces. Each interface has a different address. Thus, if a client has a reference to one interface of a given object, it cannot directly access functionality provided by the same object through a different interface. Instead, the client has

to call the method QueryInterface, the first method of any interface, with the name of the desired interface as parameter. COM aggregation makes use of this indirection. When the wrapper is asked for an interface that it does not implement itself, it forwards the QueryInterface call to the wrappee. Alternatively, it may explicitly conceal interfaces of the wrappee by answering request negatively itself instead of forwarding them. The wrappee holds a back pointer to the wrapper. When the wrappee's QueryInterface is called directly by a client, the wrappee forwards the call to the wrapper. In summary, unless the wrapper conceals part of the wrappee, aggregation satisfies the transparency requirement (3). On the negative side, aggregation only works with specially coded classes as wrappees. A programming language could enforce the rules for aggregation so that all components written in this language would be aggregable. Aggregation requires the inner object to be created along with the outer object to guarantee that only the wrapper holds a reference to the wrappee.

Ibrahim and Szyperski [27] have formalized parts of COM, including containment, aggregation, and QueryInterface. The latter is replaced by typecase statements in their exemplary language COMEL. Aiming for a truthful formalization, COMEL has the same properties as COM on the binary level.

JavaBeans In its current version, JavaBeans does not support wrappers. A very rudimentary draft proposal for an object aggregation/delegation model [8] was scrapped after public criticism. In this conventions-based approach, the wrapper (delegator) was to hold references to a number of wrappees (delegatees), but not to implement the static wrappee type. Every wrapper was supposed to implement the interface Aggregate:

```
public interface Aggregate {
  Object getInstanceOf(Class delegateInterface);
  boolean isInstanceOf(Class delegateInterface);
}
```

Instances of Class represent classes and interfaces in a running Java application. Thus, delegates could have been retrieved by naming the desired interface or class. Any object could have been wrapped, but only so-called 'cognizant' delegates would have contained a back pointer allowing the discovery of the delegator from the delegate.

Enterprise JavaBeans and CORBA Components Enterprise JavaBeans [51] and CORBA Components [40] are enterprise component standards. Their focus is on containers providing such functionality as transactions, security, events, and persistence. However, they do not provide any special support for wrappers. Like their COM siblings, CORBA components can implement multiple interfaces (facets), but only navigation within components is provided by provide_name, the rough equivalent to COM's QueryInterface.

11 Conclusions

Late composition of software components from different vendors is the essence of component software, enabling component markets and flexible reuse. One form of late composition is the combination of features implemented by different vendors into object-aggregates that appear as single objects to their clients. Our analysis shows, that existing technologies fail to fully unlock this power.

To remedy the problem, we have proposed generic wrappers, a typed form of dynamic inheritance. We have analyzed the design space with respect to both type soundness and semantic intuition, desirability, and consistency with existing mechanisms, such as subclassing. One option is forwarding instead of delegation to loosen the coupling and, thereby, avoid the semantic fragile base class problem. Another option is the snappy assignment of the wrappee to facilitate modular semantic reasoning.

As a proof of concept, we have chosen a consistent set of desirable features for a concrete mechanism, which we added to Java. We have given a mechanized proof of type soundness for the extended language. Additionally, the formalization provides an operational semantics for Java extended with generic wrappers.

Acknowledgments David von Oheimb and Tobias Nipkow provided us with their formalization of Java and helped us with our extensions. We would like to thank Ralph Back, Dominik Gruntz, Cuno Pfister, and Clemens Szyperski for a number of fruitful discussions and comments.

References

[1] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Proceedings of OOPSLA '97*, pages 49–65. ACM Press, 1997.

[2] Pierre America. Designing an object-oriented programming language with behavioral subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop*, pages 60–90. LNCS 489, Springer Verlag, 1991.

[3] D. Ancona, G. Lagorio, and E. Zucca. Jam: A smooth extension of Java with mixins. Technical report, DISI, University of Genova, 1999.

[4] Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *Proceedings of ECOOP '99*, pages 43–66. LNCS 1628, Springer Verlag, 1999.

[5] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.

[6] Kim B. Bruce, Robert van Gent, and Angela Schuett. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proceedings of ECOOP '95*, pages 27–51. LNCS 952, Springer Verlag, 1995.

[7] Martin Büchi and Wolfgang Weck. Compound types for Java. In *Proceedings of OOPSLA '98*, pages 362–373. ACM Press, 1998. http://www.abo.fi/~mbuechi/publications/OOPSLA98.html.

[8] Laurence Cable and Graham Hamilton. *A Draft Proposal for a Object Aggregation/Delegation Model for Java and JavaBeans (Version 0.5)*. Sun Microsystems, April 1997.

[9] Luca Cardelli. Type systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997.

[10] Luca Cardelli and John C. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1(1):3–48, 1991.

[11] Craig Chambers. The Cecil language: Specification & rationale (version 2.1). Technical report, University of Washington, March 1997.

[12] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of OOPSLA '98*, pages 48–64. ACM Press, 1998.

[13] William Cook. A proposal for making Eiffel type-safe. In *Proceedings of ECOOP '89*, pages 57–70. Cambridge University Press, 1989.

[14] Stephen R. Davis. *AFC Programmer's Guide*. Microsoft Press, 1998. See also http://www.microsoft.com/java/afc/.

[15] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison Wesley, 1998. http://www.catalysis.org.

[16] Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.

[17] Kathleen Fisher and John C. Mitchell. Notes on typed object-oriented programming. In *Proceeding of Theoretical Aspects of Computer Software*, pages 844–885. LNCS 789, Springer Verlag, 1994.

[18] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 171–183. ACM Press, 1998.

[19] Michael Franz. The programming language Lagoona: A fresh look at object-orientation. *Software – Concepts and Tools*, 18(1):14–26, March 1997.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[21] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[22] Mark Grand. *Patterns in Java*, volume 1. John Wiley & Sons, 1998.

[23] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proc. 18th ACM Symp. Principles of Programming Languages*, pages 131–142. ACM Press, 1991.

[24] William Harrison, Harold Ossher, and Peri Tarr. Using delegation for software and subject composition. Technical Report RC-20946 (92722), IBM Research Division, T.J. Watson Research Center, August 1997.

[25] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of OOPSLA '91*, pages 271–285. ACM Press, 1991.

[26] Urs Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings of ECOOP '93*, pages 36–56. LNCS 707, Springer Verlag, 1993.

[27] Rosziati Ibrahim and Clemens Szyperski. Can the component object model (COM) be formalized? – The formal semantics of the COMEL language. In *Proceedings of IRW/FMP'98*. Technical Report TR-CS-98-09, The Australian National University, September 1998.

[28] Ralph Keller and Urs Hölzle. Binary component adaptation. In *Proceedings of ECOOP '98*, pages 307–329. LNCS 1445, Springer Verlag, 1998.

[29] Gregor Kiczales et al. Aspect-oriented programming. In *Proceedings of ECOOP '97*, pages 220–242. LNCS 1241, Springer Verlag, 1997.

[30] Günter Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings of ECOOP '99*. LNCS 1628, Springer Verlag, 1999.

[31] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of OOPSLA '86*, pages 214–223. ACM Press, 1986.

[32] Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[33] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, second edition, 1992.

[34] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[35] Mira Mezini. Dynamic object evolution without name collisions. In *Proceedings of ECOOP '97*, pages 190–219. LNCS 1241, Springer Verlag, 1997.

[36] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *Proceedings of ECOOP '98*, pages 355–374. LNCS 1445, Springer Verlag, 1998.

[37] Anna Mikhajlova and Emil Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proceedings of FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, pages 82–101. LNCS 1313, Springer Verlag, 1997.

[38] NeXT Software, Inc. *Object-Oriented Programming and the Objective-C Language*. Addison-Wesley, 1993.

[39] Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170. ACM Press, 1998.

[40] Object Management Group. CORBA components, 1999. Revision February 15, 1999, formal document orbos/99-02-01, http://www.omg.org.

[41] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, pages 119–156. LNCS 1523, Springer Verlag, 1999.

[42] Geoff Outhred and John Potter. Extending COM's aggregation model. In *Component-Oriented Software Engineering Workshop (in conjunction with the Australian Software Engineering Conference)*, 1998.

[43] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828, Springer Verlag, 1994. http://www.cl.cam.ac.uk/Research/HVG/Isabelle/.

[44] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. 16th ACM Symp. Principles of Programming Languages*, pages 242–249. ACM Press, 1989.

[45] Dale Rogerson. *Inside COM*. Microsoft Press, 1996.

[46] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of OOPSLA '86*, pages 38–45. ACM Press, 1986.

[47] Patrick Steyaert and Wolfgang De Meuter. A marriage of class- and object-based inheritance without unwanted children. In *Proceedings of ECOOP '95*, pages 127–144. LNCS 952, Springer Verlag, 1995.

[48] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.

[49] Sun microsystems. Java platform, 1998. http://java.sun.com.

[50] Sun Microsystems, Inc. Java Beans, 1997. http://java.sun.com/beans/.

[51] Sun Microsystems, Inc. Enterprise JavaBeans, 1999. http://java.sun.com/products/ejb/.

[52] Clemens A. Szyperski. Independently extensible systems — software engineering potential and challenges. In *Proceedings of the 19th Australasian Computer Science Conference, Melbourne,* 1996.

[53] Clemens A. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[54] D. Ungar and R.B. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA '87,* pages 227–241. ACM Press, 1987. Revised version in *Lisp and Symbolic Computation,* 4(3), 187–205, 1991.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.fi

University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**