Dynamic Object Replacement and Implementation-Only Classes

Pascal Costanza
University of Bonn, Institute of Computer Science III
Römerstr. 164, D-53117 Bonn, Germany
costanza@cs.uni-bonn.de

June 27, 2001

Abstract

GILGUL is an extension of the Java programming language that allows for dynamic object replacement without consistency problems. This is possible in a semantically clean way because its model strictly separates the notions of reference and comparison that are usually subsumed in the concept of object identity. Since GILGUL's new operations respect Java's type system, objects can still be replaced only by instances of the same class or any subclasses thereof, but not by instances of any superclass. The concept of implementation-only classes is an extension of the type system that allows classes to be declared that must never be used as types. Consequently, instances of implementation-only classes can always be replaced by instances of their superclasses. This effectively widens the range of both anticipated and unanticipated adaptations.

1 The TAILOR Project

1.1 Unanticipated Adaptation

Software requirements are in a constant flux. Some changes in requirements can be anticipated by software developers, so that the necessary adaptations can be prepared for, for example by suitable parameterization or by application of dedicated design patterns. Within the scope of these anticipated options for adaptation, software can be used already in a flexible way, albeit in restricted limits und with a corresponding increase in development effort.

However, unanticipated changes of requirements occur repeatedly in practice, and the above suggested techniques cannot tackle them by definition. Furthermore, the manual integration of hooks for any conceivable eventuality is not a feasible option, since this dramatically decreases reliability, efficiency and maintainability of systems.

Alternatively, programming languages and runtime systems should be equipped with features that allow for far reaching manipulations of program internals without destructively modifying its source code. This leads to an increase of options for unanticipated adaptations as well as a decrease of effort to prepare for anticipated adaptations. Therefore the structure of programs can be kept much simpler from the outset and they can be focussed on solving their primary tasks.

In order to provide for this significant simplification of software development, the goal of the TAILOR Project at the Institute of Computer Science III of the University of Bonn [22] is to conceive and implement enhancements of programming languages and runtime systems to allow for unanticipated adaptability of software. In doing so, special attention is payed to the following issues.

1.2 Component-Oriented Software

The focus on Component-Oriented Software pushes the limits even further, since it generally is not required that components can be modified in unanticipated ways when being integrated into a system. On the contrary, components usually are deployed using a compiled format, and their source code is not available for modifications. Even if the source code can be accessed in some cases, destructive modifications are still not feasible, since they would lapse in the presence of new versions of a component.

However, it is still highly desirable to be able to use components in unanticipated contexts and with unanticipated adaptations. How can such unanticipated adaptations be carried out on a software component when only its interfaces are guaranteed to be known and when the need to update to new versions in the future have to be taken into account?

1.3 Reduction of Downtime

Essentially, unanticipated adaptations of software can take place at two points in time. They can be performed before a program is being linked into its final form, or they can happen during runtime.

Adaptations before linktime can be tackled by systems that allow for transformations of source code or similar modifications of binary representations of components [13]. Changes to software that are carried out

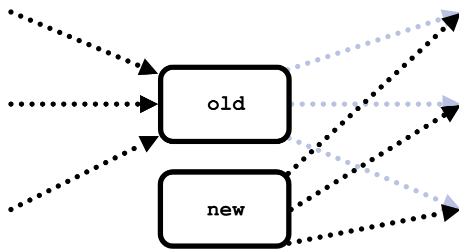

Figure 1: If an exchange of objects is executed by manual redirection of references, message sends can occur to both objects in between, probably leading to an inconsistent state.

using these systems can be made effective only by stopping an old version of a program and starting the new one. This results in downtimes that can be very long depending on the complexity of a program. These downtimes induce high costs and possibly determine an application's success or failure.

Complete elimination of downtimes can be obtained either by employing techniques that allow for anticipated adaptations during runtime (design patterns, etc.), including the restrictions and disadvantages that are discussed above. Alternatively, runtime systems must be provided with features that allow for subsequent unanticipated adaptations of an already active program.

1.4 Challenges

Accordingly, the TAILOR Project deals with enhancements of programming languages and runtime systems that allow for unanticipated adaptability of software on the stringent condition that software components are to be included whose source code is not available, and that modifications can still be made on already active programs.

Under what circumstances are such kinds of adaptations still feasible? What are the outer limits that restrict the degree of adaptability? What are the deployment costs of the envisioned enhancements? What conclusions can be drawn for the design of programming languages, runtime systems, and other software development tools?

The close examination of these questions in the course of the TAILOR Project gains insights that are valuable for the development of adaptable software in the general case as well.

2 Dynamic Replacement of Objects

In principle, unanticipated adaptation can always be dealt with by manual redirection of references. If you know the reference to an object and want to add or replace a method or change its class, you can simply assign this reference a new object with the desired properties. The new object can even reuse the old object by some form of delegation [12], so that a recovery of the old state is not needed.

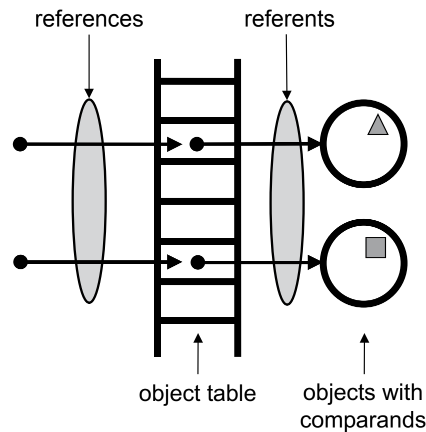

Figure 2: Identity Through Indirection — references to objects are realised as OOPs — combined with Identity Through Surrogates — each object stores a comparand.

However, there are two consistency problems involved in this approach on the conceptual level. Firstly, if there is more than one reference to an object, they all must be known to the programmer in order to consistently redirect them. Secondly, even if all references are known, they have to be redirected to the new object one by one. This approach is likely to lead to an inconsistent state of the involved objects if message sends via these references occur during the course of the redirections (for example within another thread; see figure 1).

It would be straightforward if we could simply "replace" an object by another one without changing the involved references. Such a replacement would be an atomic operation and hence, avoids the consistency problems shown above. We have discussed this subject previously on the basis of a specific example in [6]. In that paper we have also shown that a dissection of the concept of object identity and a strict separation of the included notions of reference and comparison is needed in order to introduce an operator for dynamic object replacement into a programming language.

This can be illustrated with an implementation technique called "Identity Through Indirection" in [11]: Here, a reference to an object is realised as an object-oriented pointer (OOP). An OOP points to an entry in an object table which holds the actual memory addresses. Since we do not want to restrict our model to this implementation technique, we abstract from this terminology and say that object references point to entries which hold *referents* that represent the actual objects (see figure 2).[1]

In our approach, references are never compared. To be able to compare objects we combine "Identity Through Indirection" with "Identity Through Sur-

[1] An efficient implementation scheme is outlined in section 6.

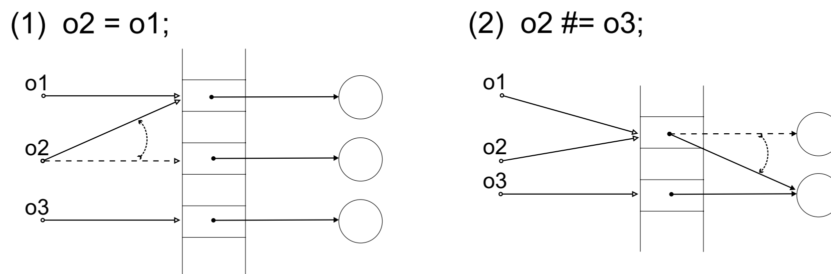

Figure 3: Referent Assignment: After execution of o2 #= o3, all three variables refer to the same object. Since o1 holds the same reference as o2, it is also affected by this operation.

rogates" [11]. Each object is supplemented with an attribute that stores a *comparand*. Comparands are system-generated, globally unique values that cannot be manipulated directly by a programmer. The comparison of objects (o1 == o2) then means the comparison of their comparands (o1.comparand == o2.comparand), but they are never used for referencing.[2]

2.1 GILGUL

Based on this scheme, we outline the programming language GILGUL in the following sections. It is an extension of Java that is currently being developed at the Institute for Computer Science III of the University of Bonn. It introduces means to manipulate referents and comparands and has been carefully designed not to compromise compatibility with existing Java sources.

There are four levels that can be manipulated when dealing with variables: the reference and the object level that already exist in Java, and the referent and the comparand level that are new in GILGUL. A class instance creation expression (new MyClass(...)) results not only in the creation of a new object, but also in the creation of a new reference, a new referent and a new comparand. The class instance creation expression returns the reference to the object's referent, which in turn has the comparand among its attributes.

2.2 Operations on Referents

In GILGUL, the *referent assignment operator* #= is introduced to enable the proposed replacement of objects.[3] The referent assignment expression o1 #= o2 lets the referent of the variable o1 refer to the object o2 without actually changing any references. Effectively, this means that all other variables which hold the same reference as o1 refer to the object o2, too. This can be realised simply by copying the referent of o2 to the entry of o1 in the object table.

Consider the following statement sequence.

```
o1 = new MyClass();
o2 = o1;
o2 #= o3;
```

After execution of the referent assignment, all three variables are guaranteed to refer to the same object o3, since after the second assignment, o1 and o2 hold the same reference (see figure 3).

Figure 3 also illustrates why a strict separation of reference and comparison is needed in order to allow for this kind of manipulations. Assume that you want to compare o2 and o3 after execution of the statement sequence given above, resulting in the scenario on the right-hand side of figure 3. In this situation, comparison of variables without the use of comparands is ambiguous on the conceptual level, since comparison of the references would yield false, whereas comparison of the referents would yield true. The decision for one or the other option would be arbitrary and cannot be justified other than by implementation-specific considerations only. Therefore, the only reasonable way to go is to opt for comparison of properties that are stored inside of the involved objects and thus make comparison of variables unambiguous.

Note that the referent assignment operator #= is a reasonable language extension due to the fact that the standard assignment operator = that is already defined in Java copies the reference from the right-hand operand to the left-hand variable, but not the referent stored in the respective entry in the object table.

Since the null literal does not refer to any object, the referent assignment is prevented from being executed on null. The expression null #= o2 is rejected by the compiler, and o1 #= o2 throws a GilgulRestrictionException when o1 holds null.[4] This ensures that a programmer is

[2] In [11] this additional attribute is named *surrogate*. Elsewhere, names like "key" and "identifier", or acronyms like "OID" and "id" are used for this concept. However, these and other terms that are found in the literature might raise the wrong associations. In our approach, we have originated the artificial word *comparand* to stress that this attribute is a passive entity that is never used for referencing, but strictly within comparison operations only.

[3] The hash symbol # is meant to resemble the graphical illustration of an object table.

[4] The GilgulRestrictionException is an unchecked exception, so this case is similar to the throw of a NullPointerException when attempting to access the properties of an object that holds null. Both kinds of exception can be avoided by testing variables against null beforehand.

not able to erroneously redirect all variables holding null to a non-null object. Note, however, that o1 #= null is valid when o1 does not hold null and redirects all variables having the same reference as o1 to null.

2.3 Operations on Comparands

It is obvious from a technical point of view that comparands may be copied freely between objects. There are, in fact, good reasons on the conceptual level to allow for copying of comparands. For example, decorator objects usually have to "take over" the comparand of the decorated object so that comparison operations that involve "direct" references to a wrapped object yield the correct result. Other usage scenarios are given in [5].

Comparands are introduced in GILGUL by means of a (final) pseudo-class java.lang.Comparand which can be used to create new comparands via class instance creation expressions (new Comparand()). By default, the definition of java.lang.Object includes an instance variable of this type, as follows.

```
public class Object {
  public Comparand comparand;
  ...
}
```

The equality operators == and != that are already defined on references in Java are redefined in GILGUL to operate on comparands, such that o1 == o2 means the same as o1.comparand == o2.comparand, and o1 != o2 means the same as o1.comparand != o2.comparand.

Given these prerequisites, we can let a wrapper "take over" the comparand of a wrapped object in order to make them become equal by simply copying it as follows: o1.comparand = o2.comparand.

Ensuring the uniqueness of a single object is always possible by assigning a freshly created comparand as follows: o1.comparand = new Comparand().

The comparand of the null literal is prevented from being accessed via null.comparand, or o1.comparand when o1 holds null, so it cannot be copied to other objects, and it cannot be replaced. An attempt to access null.comparand is rejected by the compiler, and o1.comparand throws a GilgulRestrictionException when o1 holds null. This ensures that testing equality against null is guaranteed to be unambiguous.

Since comparands cannot be manipulated directly, there are no limitations on how they are implemented in a concrete virtual machine. The only requirement they have to fulfil is that if o1.comparand and o2.comparand have been generated by the same (different) class instance or comparand creation expression, then o1.comparand == o2.comparand yields true (false), and o1.comparand != o2.comparand yields false (true).

For example, a reasonable and efficient implementation of comparands are 64-bit unsigned integers with comparand creation being accomplished by increment of a global counter.[5] This scheme provides for approximately 10 billion unique comparands per second for half of a century.[6]

Still, the actual implementation of comparands is hidden from programmers. Especially, GILGUL prevents comparands from being cast to any other type and, for example, does not allow arithmetic operators to be executed on comparands.

2.4 Operations on References and Objects

Besides GILGUL's new operations on referents and comparands, the operations on references and objects are still available as a matter of course. However, there are some interdependencies between the standard methods equals(...) and hashCode() and the ability to copy comparands between objects. As a consequence, the standard definition of hashCode() has been changed to return a hash code value for an object's comparand. More details on this subject are given in [6].

3 Declaration of Restrictions

GILGUL offers the possibility of declaring restrictions on what operations are valid on concrete referents and comparands. Since references and comparands are created at the same time as their initially corresponding objects via class instance creation expressions, these restrictions have to be given in constructor declarations as follows.

```
class C {
  public C()[#fixed, bound] {...}
}
```

The possible restrictions are fixed, bound or none for comparands, and #fixed, #bound or none for referents.

Restricitions on Comparands If no restriction is declared for a comparand, it may be copied or replaced freely. If a comparand is declared as fixed, it cannot be replaced by another comparand, but it may be copied elsewhere. If a comparand is declared as bound, it may neither be replaced nor copied, which means that a bound comparand is implicitly fixed.

The rationale behind this implication is that if a programmer declares a comparand as bound, he/she wants to guarantee that there does not exist a copy of this comparand elsewhere. However, if a bound comparand could be replaced by a comparand of another object, this guarantee would be violated, because the other object could not be prevented from using the latter comparand.

[5] This must be synchronized in a multi-threaded environment, since write accesses to 64-bit values are usually not atomic.

[6] On the other hand, 32-bit values are usually not big enough to ensure uniqueness for long-running applications. At a rate of 1000 comparands per second, they wrap around after roughly 6 weeks.

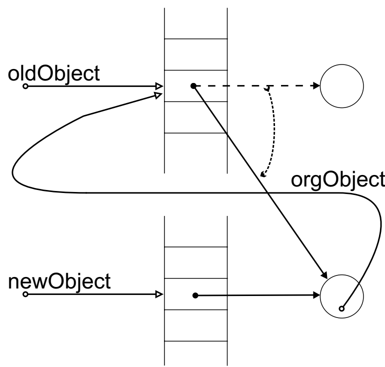

Figure 4: Naive application of oldObject #= newObject results in an unwanted cycle: When newObject.orgObject holds the same reference as oldObject beforehand, it will refer to newObject afterwards.

Restrictions on Referents Similarly, if no restriction is declared for a referent, it may be copied or replaced freely. If a referent is declared as #fixed, it cannot be replaced by another one, but it may be copied elsewhere. If a referent is declared as #bound, it may neither be replaced nor copied, which means that a #bound referent is implicitly #fixed. The rationale is the same as for comparands.

These constructs allow one to flexibly declare detailed restrictions on comparands and referents, ranging from the allowance of all operations introduced in the previous sections to the reduction to the "classic" approach of dealing with object identity. For example, consider the following class declaration.

```
class C {
  public C()[bound, #bound] {...}
}
```

Instances of this class can neither have their referents nor comparands replaced nor copied.

Note that in contrast to the standard access modifiers of Java (public, protected, private), these restrictions on comparands and referents are not attached to variables and consequently cannot be checked statically in the general case. Therefore, attempts to access comparands via o1.comparand and referent assignments (o1 #= o2) may throw instances of GilgulRestrictionException. The restrictions imposed on the null literal in sections 2.2 and 2.3 can be restated as if null's "constructor" had been declared as [bound, #fixed], so the presumed exceptional cases for null are direct consequences of this "declaration".

Further note that GILGUL's flexibility comes at the price of an increased complexity of contracts, since it still must be determined which kinds of referent assignment and comparand assignment are valid for concrete classes or objects. The possible restrictions on comparands and referents just help to make these contracts more explicit, but they do not reduce their complexity at the conceptual level, as is the case for the standard access modifiers [14].

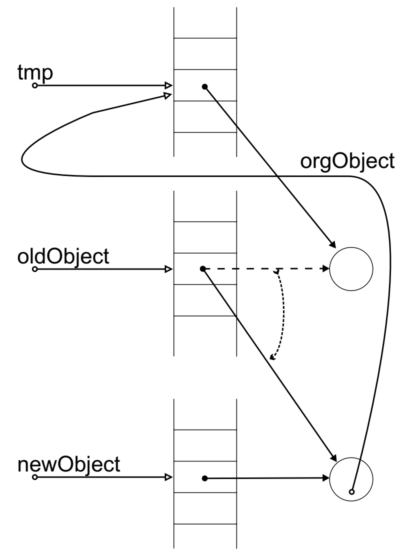

Figure 5: Correct application of oldObject #= newObject: When newObject.orgObject holds a different reference to the same object as oldObject beforehand, it will still refer to the former oldObject afterwards, since the temporary reference is not affected. Therefore, the unwanted cycle is avoided.

3.1 Example of Use

Returning to our given problem, we are now able to apply the new operations to achieve the desired replacement of an object atomically. We can apply oldObject #= newObject to let newObject replace oldObject consistently for all clients that have references to oldObject.

However, one has to be careful when newObject wants to refer to oldObject in order to delegate messages that it cannot handle by itself. Regard the following naive sequence of operations.

```
newObject.orgObject = oldObject;
oldObject #= newObject;
```

This would be erroneous, because afterwards newObject.orgObject would refer to newObject, since it contains the same reference as oldObject according to the first assignment. This, of course, leads to a cycle and therefore, to non-terminating loops for messages that cannot be handled by newObject (see figure 4). The following statement sequence however is correct (see figure 5).

```
// let a new reference refer to oldObject
   tmp #= oldObject;

// use tmp instead of
// oldObject for forwarding
   newObject.orgObject = tmp;
```

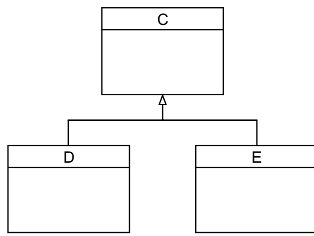

Figure 6: One might want to replace an instance of class D by instances of class C or E.

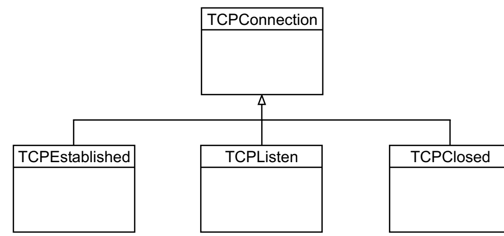

Figure 7: An example of the State Pattern. Only TCPConnection is used as a type.

```
// ensure that equality behaves well
   newObject.comparand
                    = oldObject.comparand;

// tmp and so newObject.orgObject
// remain unchanged
   oldObject #= newObject;
```

The actual "replacement" of oldObject is initiated by the last operation, and thus is indeed atomic. Further note that the temporary variable can be used to revert the replacement by application of oldObject #= tmp.

However, this idiom is only needed when newObjects needs to reuse oldObject. Otherwise, a "simple" replacement is sufficient. In this case, reversal of a replacement can also be achieved by the use of an additional reference, but it is not needed for forwarding purposes.

As we can see, in all of these variants, GILGUL's new operations give the programmer the possibility to "replace" the former object atomically and thereby relieves him/her from having to deal with any consistency problems. Furthermore, the involved objects need not anticipate such modifications, reducing the complexity of the development of actual components to a great extent.

4 Type Issues

The referent assignment operator #= respects Java's type system. In the following sections we explore what this actually means and especially show that this may lead to unnecessarily restricted situations. An extension of Java's type system is presented afterwards that allows pure implementation classes to be declared that do not define new types. This is a novel approach, which to our knowledge is not available in any previous programming language. This extension allows the restrictions to be resolved, which are associated with the use of the referent assignment operator.

4.1 Additive and Subtractive Replacement

The referent assignment operator allows an object to be replaced always by another one that implements at least the same types (*additive replacement*). A special case is the replacement by an object that is an instance of exactly the same class, but this statement also includes objects that are instances of any of the old object's subclasses. This is a direct consequence of the Liskov Substitution Principle [19].

The situation becomes more complex when you want to replace an object that implements an unrelated type. For example, given the class hierarchy of figure 6, you might want to replace an instance of class D by an instance of class C or E (*subtractive replacement*). This situation for example occurs in a variation of the State Pattern [8], where these classes represent states and a change of state is expressed by a replacement of a state object. Assume that you want to avoid the use of forwarding in the implementation of the State Pattern, but instead want to consistently express state changes for all clients that refer to a state by application of the referent assignment operator.

In this and similar situations, before replacement of an object by another one, it must checked dynamically that the old object is referenced only by variables that expect it to implement the intersection of the types implemented by both the old and the new object. Therefore, the runtime system has to keep track of all references to an object and their respective types. However, in order to ensure that subtractive replacements are always possible whenever needed, an extension of Java's type system is needed.

4.2 Implementation-Only Classes

On closer examination of an example application of the State Pattern, as depicted in figure 7, it can be noticed that it is likely that classes TCPEstablished, TCPListen and TCPClosed are never needed as actual types of their own. Instead, all clients that are in need of a TCPConnection always use this general class type. This fact can be expressed explicitly in GILGUL by adding the modifier implementationonly to the classes that are not needed as types (all except for TCPConnection) as follows.

```
implementationonly class TCPEstablished
                        extends TCPConnection
{...}
```

Afterwards, these classes can still be used as any other class in most respects, for example within instance creation expressions. However, they cannot be used as types

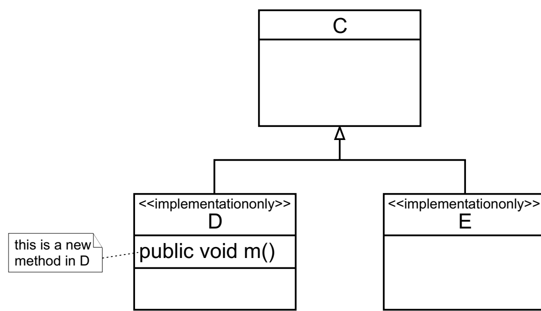

Figure 8: How can a new method in an implementation-only class be called from the outside?

anymore, in the sense that variables must not be declared as being of a type of an implementation-only class. Consequently, instances of these classes (TCPEstablished, etc.) can always be replaced by instances of any class of the given hierarchy, since they all implement the same set of types (which consists of the type of TCPConnection) by definition. This property results in the desired applicability of subtractive replacement.

4.3 Cast Expressions

There is still a restriction involved in the declaration of implementation-only classes that has not been addressed yet. Assume that in figure 6 classes D and E are declared as implementation-only classes, but for example D additionally defines a public method m() which is not defined in the other classes C and E (see figure 8).

How can this method ever be called from outside of class D? The only way instances of class D can be used is via references of type C as follows: C obj = new D().

Sending the message m() to obj would result in a compile-time error, since m() is not included in the interface of C.

For this reason, GILGUL makes an exception from the general rule that implementation-only classes must never be used as types in the case of cast expressions. Therefore, message m() can be sent to obj by casting obj to class D beforehand, as follows: ((D)obj).m().

This exceptional case does not conflict with the original goal of implementation-only classes, that is to allow for subtractive object replacement. Variables which are cast to an implementation-only class still cannot be assigned to variables that are declared to be of this very class type, because the restriction still holds that it must not be used as a type. Therefore, in the very moment of applying the referent assignment operator to an instance of an implementation-only class, it is still ensured that there are no variables in the running system that expect the new object to implement the old object's class type.

4.4 The with Statement

In order to conveniently express cascaded method calls to the same variable in the presence of implementation-only classes, GILGUL introduces a with statement that is reminiscent of the similar statement in the programming language Oberon [16]. In GILGUL, it can be used as follows: with (obj instanceof D) {...}.

Its effect is that obj is regarded as an instance of the respective class for the scope of the following (block) statement. The type given in the with condition can be any interface or class type, including the type of an implementation-only class. (This is the second and last exception from the rule that implementation-only classes must never be used as types.) Given this statement, a call to a method defined in an implementation-only class can be expressed as follows.

```
with (obj instanceof D) {
   obj.m();
   ...
   obj.otherNewMethodsInD();
}
```

Note that whereas in Oberon the with statement is introduced to allow for compiler optimization, such that code for the with condition is emitted only once, in GILGUL the with statement is syntactic sugar only. Since the object referred to by the variable in the with condition can always be replaced by another object (for example within another thread), the condition might not hold for the following block completely, possibly resulting in a class cast exception at any place within that block where the variable is actually used.

4.5 Relation to Java's Interfaces

Apart from the confined use of implementation-only classes as types, they do not differ from usual classes. Especially, they are allowed to implement any interface, as follows.

```
implementationonly class C
                  extends D implements I
{...}
```

Note, however, that this declaration introduces a new type into the class hierarchy if interface I is not implemented by any of C's superclasses. As soon as a variable of type I refers to an instance of class C, this instance can be replaced only by objects that simultaneously are instances of any subclass of D and implement I.

Yet, we have not chosen to disallow implementation-only classes to implement interfaces, since this feature can be utilized for a clean separation of types and implementations as follows.

```
interface I {...}

implementationonly class C implements I
// Note that C has no superclass.
{...}
```

Since the main purpose of Java's concept of interfaces is the declaration of types, we also have not chosen to allow for some kind of "implementation-only interfaces" (or better: "optional interfaces"). This could have been useful in order to group related methods into such interfaces without declaring new types, which would be similar to Smalltalk's concept of categories. This also could have helped to avoid the declaration of new types just to introduce application-wide constants. However, the first use is only of marginal value, and the second stretches an unfortunately widely adopted abuse of Java's interfaces too far.

4.6 Relation to Unanticipated Adaptation

The introduction of implementation-only classes fits perfectly to the goal of widening the range of unanticipated adaptations. It is always possible to extend an existing class hierarchy by additional implementation-only classes and thus allow for both additive and subtractive object replacements. There is no need to change existing classes, so this feature can be used for third-party components without further effort.

The possibility to introduce new methods into an implementation-only class without restricting the replaceability of its instances also improves adaptability. Since implementation-only classes can be used as types in cast expressions, these new methods can be called within unrelated classes that yet are aware of these new methods. Still, the existing class hierarchy does not need to be changed for this purpose.

The fact that casts can be checked dynamically only, and therefore might raise class cast exceptions, may be regarded as a disadvantage of this proposal. However, this is only the flipside of the possibility to declare *optional methods*, which in turn is a benefit that otherwise cannot be expressed easily. Implementation-only classes effictively decouple declarations of optional properties on the one hand, that virtually can be added to and removed from an object, and the actual use of such optional properties on the other hand, which of course may result in the temporary absence of these properties.

Other approaches that allow for optional properties insist on their introduction into the existing class hierarchy in order to allow for compile-time checks of the sound use of these properties, but in this way they simultaneously narrow the range of unanticipated adaptations of third-party components. See for example the concept of *empty methods* in Component Pascal [21] which are similar to abstract methods but default to empty method bodies in order to avoid exceptions during runtime.

Implementation-only classes are not only useful in conjunction with referent assignments, but also with other programming language constructs, like Generic Wrappers [3] or Delegation [12]. For example, Generic Wrappers could be enabled to dynamically change their wrappees to instances of the wrappee's superclass, if the wrappee is an instance of an implementation-only class. Currently, Generic Wrappers do not allow for the subtractive exchange of wrappees.

5 Related Work

GILGUL is the first approach known to the author that strictly and cleanly separates the notions of reference and comparison on the level of a programming language. An overview of related work centered around the theme of object identity is given in [6].

Note that this separation of object identity concerns is orthogonal to the usual distinction between reference semantics and value semantics that is found for example in Smalltalk [18], Eiffel [15], and Java [1]. These languages allow programmers to choose from these semantics when comparing variables, but they all still rely on comparison of references in order to establish object identity.[7] For this reason, object identity usually coincides with reference semantics. In contrast, our approach relies on comparands to determine (logical) identity, and this opens up new degrees of flexibility.

Microsoft's component object model COM [2] separates object identity and references to the degree that it generally does not require components to return the same reference each time when a specific interface is requested. However, in order to enable determination of object identity by comparison of the special IUnknown reference, it is required that it must never change as an exceptional case. Again, object identity and reference semantics coincide and therefore, COM components cannot be replaced easily during runtime, but only if they explicitly are prepared for dynamic replacement. COM's *monikers* provide for an alternative mechanism of object identification that allow the physical implementation of objects to change occasionally. However, monikers are used primarily as a basis for linking and persistence in COM, but not as a general reference mechanism. After a client has bound to an object that is identified by a moniker, this object is subsequently accessed via standard references (including IUnknown).

The goal of separating interfaces and classes has been proposed explicitly first by Cook et al. [4] and, for example, has been addressed in Emerald [10], Sather [20] and Java [1]. However, whereas it is possible to declare pure interfaces/types in one or the other way in all of these approaches, the declaration of classes still implies the accompanying (implicit or explicit) declaration of interfaces in order to use newly declared properties from the outside. In GILGUL, it is possible to declare pure implementation classes that must not be used as types except within cast expressions. In this way, GILGUL "completes" the separation of types and classes that has been initiated with the former approaches.

[7] Whether the actual semantics of comparison is defined in the respective classes or must be determined by choosing from different comparison operators/methods varies from language to language. A thorough examination of this issue is given in [9].

Newer approaches that head for subtractive object replacement, and modify the type system for this purpose in a similar way, are Fickle [7] and Wide Classes [17]. However, these approaches still do not allow for declaration of classes that must not be used as types.

6 Conclusions and Future Work

We have designed the programming language GILGUL, a compatible extension to Java. It introduces the pseudo-class Comparand and the referent assignment operator #=. It also changes the definition of the existing equality operators == and != according to the GILGUL model.

This model is a generalization of what can be expressed in terms of object identity in current object-oriented programming languages. It allows one to flexibly declare detailed restrictions on the object level as needed, ranging from the unrestricted applicability of GILGUL's new operations to the reduction to the traditional stringent restrictions placed on object identity.

We have shown how GILGUL's new operations can be applied for the purpose of dynamic object replacement without the need to deal with consistency problems. We have investigated restrictions that result from the type soundness of the referent assignment operator. We have then proposed an extension of Java's type system that allows pure implementation classes to be declared that must not be used as types. This novel approach "completes" previous efforts to separate pure interfaces from classes that, however, still define types of their own. It effectively widens the range of both anticipated and unanticipated adaptations.

Currently, a compiler and runtime system for GILGUL is being developed at the University of Bonn. We are modifying a concrete implementation of the Java Virtual Machine and attempt to include optimizations, for example the use of direct pointers to an object as long as possible until an extra level of indirection is inevitable, and the subsequent reversal to direct pointers as soon as this extra level has become unnecessary. Future work also includes a formal proof of type soundness.

An issue we have not discussed in this paper is the semantics of dynamic object replacement in the presence of methods that simultaneously execute on the target object. Essentially, if the active method is part of the same thread as the attempt at replacement an execption is thrown, and if these actions are part of different threads they are synchronized accordingly. Details will be reported elsewhere.

6.1 Acknowledgements

The author thanks Tom Arbuckle, Michael Austermann, Peter Grogono, Arno Haase, Günter Kniesel, Thomas Kühne, Sven Müller, James Noble, Markku Sakkinen, Oliver Stiemerling, Dirk Theisen, Kris De Volder and many anonymous reviewers for their critical comments on earlier drafts and related publications.

This work is located in the TAILOR project at the Institute of Computer Science III of the University of Bonn. The TAILOR project is directed by Armin B. Cremers and supported by Deutsche Forschungsgemeinschaft (DFG) under grant CR 65/13.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language, Second Edition*. Addison-Wesley, 1998.
[2] D. Box. *Essential COM*. Addison-Wesley, 1998.
[3] M. Büchi and W. Weck. *Generic Wrappers*. in: *ECOOP 2000*. Proceedings, Springer.
[4] W. R. Cook, W. C. Hill, and P. S. Canning. *Inheritance is not subtyping*. in: *POPL '90*. Proceedings, ACM Press.
[5] P. Costanza and A. Haase. *The Comparand Pattern*. accepted for *EuroPLoP 2001*, Irsee, Germany.
[6] P. Costanza, O. Stiemerling, and A. B. Cremers. *Object Identity and Dynamic Recomposition of Components*. in: *TOOLS Europe 2001*. Proceedings, IEEE Computer Society Press.
[7] S. Drossopoulou, F. Damiani, M. Dezani, and P. Giannini. *Fickle: Dynamic Object Reclassification*. in: *ECOOP 2001*. Proceedings, Springer.
[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
[9] P. Grogono and M. Sakkinen. *Copying and Comparing: Problems and Solutions*. in: *ECOOP 2000*. Proceedings, Springer.
[10] E. Jul, R. K. Raj, E. D. Tempero, H. M. Levy, A. P. Black, and N. C. Hutchinson. *Emerald: A General-Purpose Programming Language*. Software – Practice and Experience, 1991, 21(1):91-118.
[11] S. N. Khoshafian and G. P. Copeland. *Object Identity*. in: *OOPSLA '86*. Proceedings, ACM Press.
[12] G. Kniesel. *Type-Safe Delegation for Run-Time Component Adaptation*. in: *ECOOP '99*. Proceedings, Springer.
[13] G. Kniesel, P. Costanza, and M. Austermann. *JMangler — A Framework for Load-Time Transformation of Java Class Files*. accepted for *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, Florence, Italy, 2001.
[14] G. Kniesel and D. Theisen. *JAC — Access Right Based Encapsulation in Java*. Software – Practice and Experience, 2001, 31(6):555-576.
[15] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
[16] M. Reiser and N. Wirth. *Programming in Oberon – Steps Beyond Pascal and Modula*. Addison-Wesley, 1992.
[17] M. Serrano. *Wide Classes*. in: *ECOOP '99*. Proceedings, Springer.
[18] D. N. Smith. *Smalltalk FAQ*. http://www.dnsmith.com/SmallFAQ/, 1995.
[19] B. Liskov. *Data Abstraction and Hierarchy*. ACM SIGPLAN Notices 23, 5, May, 1988.
[20] C. Szyperski, S. Omohundro, S. Murer. *Engineering a Programming Language: The Type and Class System of Sather*. Technical Report TR-93-064. The International Computer Science Institute, Berkeley, CA, USA, 1993.
[21] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
[22] The Tailor Project. http://javalab.cs.uni-bonn.de/research/tailor/