Object Identity and Dynamic Recomposition of Components

Pascal Costanza, Oliver Stiemerling, and Armin B. Cremers
University of Bonn, Römerstraße 164
D-53117 Bonn, Germany
{*costanza, os, abc*}*@cs.uni-bonn.de*

Abstract

Dynamic recomposition of components in a program imposes advanced requirements on the expressive power of object-oriented programming languages. For example, the replacement of a component with another reveals consistency problems stemming from the fact that the concept of object identity tries to fulfil the distinct purposes of reference and comparison. By clearly separating the two notions and providing means to manipulate them independently, the consistency problems can completely be avoided.

1: Introduction

The purpose of this paper is to demonstrate how consistency problems arising from dynamic recomposition of software components can be solved by a novel approach of dealing with object identity in an object-oriented programming language.

To set the stage, we first describe the Evolve runtime and tailoring platform as an example for a system that provides facilities for dynamic recomposition of software components. Using the simple example of inserting a new component into an Evolve application during runtime, we identify and discuss a number of consistency problems.

These problems stem from the fact that the concept of object identity combines two distinct notions. One is the facility of object reference which permits object correlation and access to objects' properties. The other is the facility of object comparison which permits one to decide if two variables refer to the same object. Both notions impose several restrictions on what can be expressed in terms of identity, and simultaneously satisfying them has resulted in the concept of object identity fulfilling only the somewhat limited intersection of these two sets of requirements.

By clearly differentiating the two notions of reference and comparison and providing means to manipulate them independently, we are able to relax the requirements for each notion and introduce new and powerful operations. By employing these operations we can completely avoid the shown conistency problems.

The following section gives an overview of the Evolve project, its runtime and tailoring platform, the FlexiBeans component model and an example application. The section discusses how component-based Evolve applications can be reconfigured during runtime to meet changing requirements. The problems arising from dynamic recomposition and object identity are discussed. Section 3 describes the consequences of separating the notions of reference and comparison, especially with regard to the possibility of introducing new

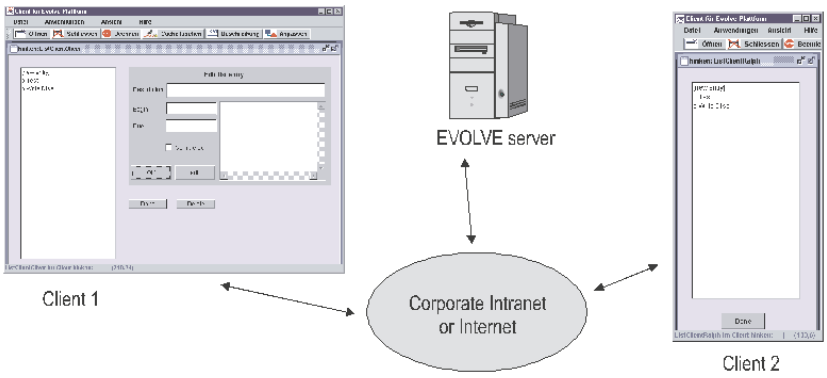


Figure 1. A simple distributed shared to-do list application based on the EVOLVE platform

operations into the Java programming language, and it discusses how the dynamic recomposition problems can be solved by these new operations. Finally, we relate our approach to other work and conclude.

2: The EVOLVE Project

The EVOLVE project [9, 10, 11] at the Institute of Computer Science III at the University of Bonn investigates the use of software components after initial development and deployment in order to provide tailorability for complex distributed applications.

In the course of the project, a runtime and tailoring platform has been designed and implemented that supports dynamic recomposition of distributed applications. This platform, its component model FLEXIBEANS and an example application are discussed.

2.1: The Evolve Platform

The EVOLVE platform is designed to support the deployment and subsequent adaptation of arbitrary distributed component-based multi-user applications. It is independent from domain specific functionality and can thus be used to provide many different software systems with the property of adaptability. Fig. 1 depicts a simple example of a component-based application made adaptable by the EVOLVE platform - a shared to-do list employed by two users to coordinate their tasks:

Client 1 belongs to a manager, client 2 to a subordinate. The actual data of the shared to-do list is stored on the EVOLVE server (in the middle of fig. 1) employing a (invisible) server component. The clients are tailored to meet the requirements of their respective owners. While the subordinate may only see the contents of the list and mark entries as "done", the manager can actually add new entries to the list and delete them. The distributed application is built from a set of prefabricated components.

In a traditional system, the composition would be static after development and deployment. The EVOLVE platform, however, maintains - and permits the manipulation of - the system's component structure. During the use of the system, a system administrator, outside consultant or even an end user can switch to the *tailoring mode* (fig. 2) in which he

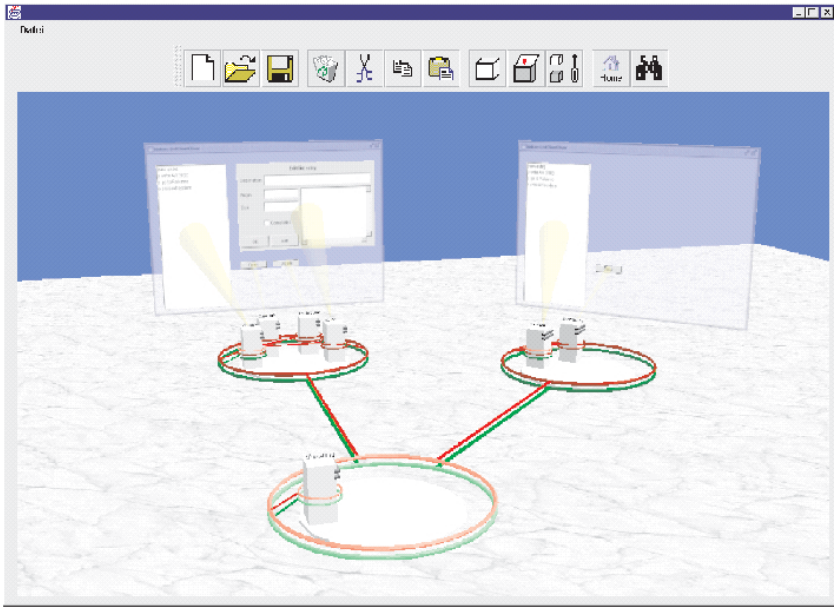


Figure 2. Screenshot of the 3D tailoring interface (showing the tailoring perspective of an application)

or she can inspect and manipulate the entire distributed application (if he or she has the right to do so).

Fig. 2 depicts a 3D user interface for component-based tailoring which accesses the flexibility provided by the EVOLVE system. The two circles in the background represent the two clients, together with the virtual screens onto which appearance and position of the visible components are projected. All components — visible and invisible — are represented as boxes depicting the compositional structure of the application. The circle in the foreground represents the server that contains the invisible component for storing the contents of the shared to-do list.

The tailor can move around the 3D component scene and perform manipulations, e.g. concerning the positioning of the visible components on the screen or the connections of the components. He can remove component instances or add new ones from a repository (not shown in fig. 2). The parameterization of component instances can also be manipulated. In short, the tailor has full control over every aspect of the application's composition, whenever the requirements of the supported group change. Furthermore,

- *tailoring operations can be performed during runtime.* In extremis, a system administrator can add components, while the user is working with the shared to-do list. The system does not have to be shut down and the state of all other components is maintained.
- *tailoring operations can be performed remotely.* The whole system can be tailored from any workstation in the network. This feature supports models of centralized and decentralized system management.
- *the effect of tailoring operations can be shared among many users.* In the example, if other users share the definition of the subordinate client, the effect of the tailoring operations is propagated to all running instances of that definition.

- *the effect of tailoring operations can be restricted to subgroups of users.* This feature permits the accommodation of individual differences.
- *tailoring operations can be applied to any level of the compositional hierarchy.* A system administrator might inspect and manipulate the system on a very fine-grained level, while an end user might prefer a more high level view.

Summarizing, the EVOLVE platform is responsible for maintaining component structures during runtime and for providing functionality for manipulating these structures. The second central element of the approach is the component model, i.e. the specification that tells the programmer exactly what a component is.

2.2: FlexiBeans

The component model of the EVOLVE platform is the FLEXIBEANS model that extends the JAVABEANS model [3] with the concepts of:

- *named ports,* permitting the differentiated event handling on the compositional level mentioned above, without having to dynamically produce and compile (adapter-)code.
- *shared objects,* permitting a less strongly synchronized style of interaction in the fashion of a "pull"-like data flow (need- and not creation-driven exchange of data).
- *remote interaction,* permitting the composition of distributed groupware applications based on Java RMI (Remote Method Invocation). A button situated on one machine can, for instance be directly connected to a component on another machine.

The complete FLEXIBEANS specification can be found in [12].

2.3: An Example Application

This section gives an example how a distributed groupware application can be built from a set of FLEXIBEANS components. In the graphical representation, shared object ports are represented by rectangles, while event ports are represented by circles. The polarity of the port (provided or required) is given by the interior of the shape (filled = provided).

Shared to-do lists support coordination of work activities in small groups (2-10 persons). They contain entries that describe a task to be done, its title, begin, deadline, and a flag indicating the status of completion (in progress, completed). Depending on the structure of the group and its work habits, there can be distinct group members who add tasks, perform tasks, check for completion, and monitor or distribute work.

The shared to-do list application presented here is highly simplified for presentation purposes. However, applications with the same basic functionality are used in IT support departments, call centers, and generally for coordination in small groups which are confronted with a lot of short-term, well-defined tasks.

The shared to-do list component set consists of four visible and one invisible component. Table 1 gives an overview and informally describes the semantics of each component. These components can now be used to compose a distributed shared to-do list application. It is the same application as in fig. 1 and 2.

This application is distributed over three locations. The *shared list* component instance resides on a server and is connected to an instance of the *supervisor client* on one machine and to an instance of the *subordinate client* on another machine. The different requirements

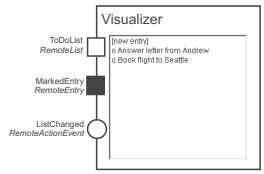
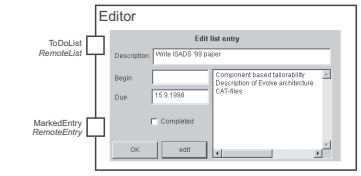
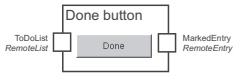
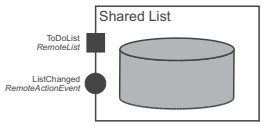
	**The *visualizer* component** This component is visible for the user and displays the current contents of a shared list. It has three ports: the first port connects to a shared list component, the second port receives events which indicate a change in a shared list, and the third port shares the currently marked entry in the list with any interested component.
	**The *editor* component** This component is actually an complex component which is composed of several visual subcomponents. For simplicity it is regarded as atomic here. The two ports connect to a shared list and the marked entry of a visualizer component. The user can add new entries to the list (if the content of marked entry is [new entry]) or edit other selected entries in the list. The large text box on the right can be used to describe the task.
	**The *delete button* component** This component is connected to a shared list and a marked entry and – if pressed – deletes the marked entry in the list
	**The *done button* component** This component is connected like the delete button. When pressed, it sets the flag of the marked entry to "completed".
	**The *shared list* component** This component is the only invisible component. It usually resides on a server and maintains a list of tasks. The list is shared via the ToDoList port and other components are notified of changes via the ListChanged event port.

Table 1. Components of the Shared To-Do List Framework

of supervisor and subordinate are met by different compositions of components taken from the set of table 1.

2.4: Problems with Dynamic Tailoring

Assume that the application described above is to be extended with a security component on the client side. This component encrypts the traffic between the client and the server.[1]

Fig. 3 shows this change. The security component is inserted between the remote connection to the server and the delete button, the visualizer and the editor component. Before we can discuss the problems that can arise when inserting the security component into the running application, we have to leave the rather abstract component level and deal with the object-oriented implementation of the FlexiBeans component model.

The shared object connection between the three client components (editor, visualizer and delete button) and the shared list component on the remote server is realised on the oo-level by giving the reference of a local stub object (on the client) representing the shared list on the remote computer to these three components. We have used Java RMI to abstract from the details of the remote interaction and will not discuss this point further in this paper.

The right side of the fig. 3 depicts how the change described above is reflected on the oo-level. In our current implementation of the Evolve platform in JDK 1.2, the change comprises the following steps (not necessary in that exact order): First the new security component has to be instantiated, then it has to be connected to the stub object. The connection of the three visible components have to be — one after the other — redirected

[1] Obviously, a similar component has to be instantiated on the server side — however, for the purpose of this section, we are only concerned with the client side.

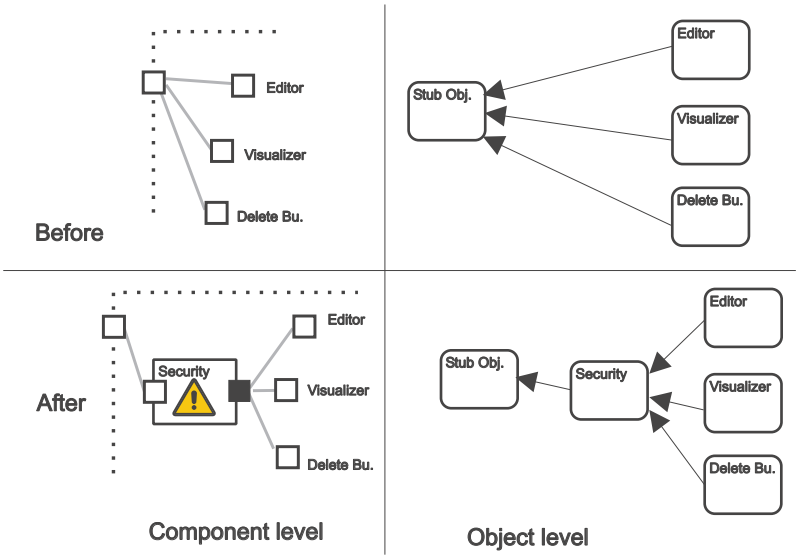


Figure 3. The supervisor client is extended with a security component (left). On the object level (right), this causes the redirection of several object references.

from the stub object to the new security object instance.[2]

The point is, that the change currently cannot be performed atomically. This causes the problem that during the tailoring operation (or rather transaction), the application is in an inconsistent state, in which, for instance, the editor component is not connected to the stub object. If the user desires to write an entry into the shared list and presses the "ok" button, nothing happens. The same problem holds, if the delete button is not connected. If the visualizer is not connected and an update event is sent from the server, the visualizer cannot show the current list and is consequently not consistent with the list on the server.

Obviously, these problems could be handled programmatically. One could implement the components in a fashion which deals in a sensible way with inconsistent connections and states. This, however, would make the components larger and — even worse — implies that the component programmer has to anticipate the user of the components in a dynamic environment. This is, at least today, almost never the case.

Consequently, we have to look for other ways to deal with unanticipated dynamic reconfigurations in a sensible and - foremost - safe way.

3: Reference and Comparison

Instead of changing each reference to the stub object and letting it refer to the new security object one by one, it would be straightforward if we could simply "replace" the stub object by the security object without changing the involved references. Such a replacement would be an atomic operation, and hence, avoids the problems shown above.

[2]These steps can, for instance, be performed in the 3D interface shown in fig. 2. The change could also be performed automatically by a small script or procedure.

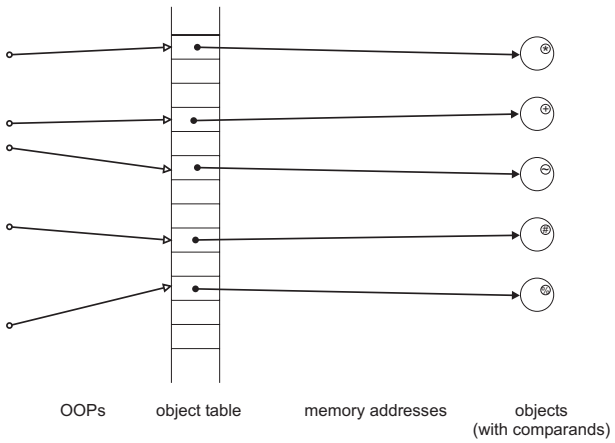


Figure 4. Identity Through Indirection — references to objects are realised as OOPs — combined with Identity Through Surrogates — each object is supplemented with a comparand.

In fact, the programming language Smalltalk provides an operation become:, that enables the programmer to "swap" the objects pointed to by two references without actually changing the references. However, the effects of become: are usually obscure and so its application is generally considered dangerous.[3]

We believe that the reason for these obscurities is that the concept of object identity in fact combines two distinct notions. One is the facility of object reference, which permits object correlation and access to objects' internal states. The other is the facility of object comparison, which permits the decision if two variables actually point to the same object. In the following paragraphs we discuss the central idea of our approach of separating the notions of reference and comparison and some of its consequences.

Our approach can be illustrated with an implementation technique called "Identity Through Indirection" in [5]: 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 physical memory addresses. This technique is employed in many Smalltalk implementations as well as the first Java implementations, among others.[4]

In our approach, the sole purpose of an OOP is to reference an object — OOPs are never compared. To be able to compare objects we combine "Identity Through Indirection" with "Identity Through Surrogates" [5]: Each object is supplemented with an attribute storing a *comparand*.[5] Comparands are system-generated, globally unique values that cannot be directly manipulated by a programmer. Comparing objects (o1 == o2) then means comparing their comparands (o1.comparand == o2.comparand), but they are never used for referencing (see fig. 4).

Based on this scheme, we outline the programming language GILGUL in the following sections. It is an extension to the Java programming language [1] that is currently being developed at the Institute for Computer Science III of the University of Bonn. It introduces

[3]For example, see the Smalltalk FAQ [8] for further details.

[4]Note that an actual implementation of our model does not have to resemble its illustration given here.

[5]In [5] the term *surrogate* is actually used for this additional attribute. Since this term might raise the wrong associations, we use the term *comparand* instead to stress that this attribute is intended to be utilized within comparison operations only.

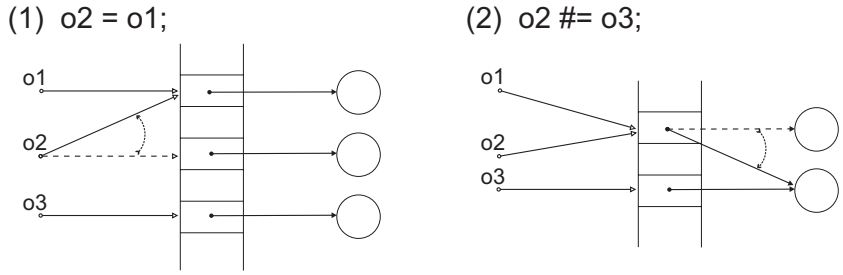


Figure 5. Reference Assignment: After executing o2 #= o3**, all three variables refer to the same object. Since** o1 **holds the same reference as** o2**, it is also affected by this operation.**

means to manipulate references and comparands and has been carefully designed not to compromise compatibility with existing Java sources.

There are three levels that can be manipulated when dealing with variables in GILGUL: the reference level, the object level and the comparand level. 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 and a new comparand. The class instance creation expression returns the reference to the object, which in turn has the comparand among its attributes.

3.1: Operations on References

In GILGUL, the reference assignment operator #= is introduced to enable the proposed replacement of objects:[6] The reference assignment expression o1 #= o2 lets the reference stored in the variable o1 refer to the object o2 without actually changing the reference. Effectively, this means that all other variables holding the same reference as o1 refer to the object o2, too. This can be simply realised by copying the memory address of o2 to the entry of o1 in the object table.

Consider the following statement sequence.

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

After executing this statement sequence, 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 fig. 5).

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

Since the null literal null does not refer to any object, the reference 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.[7] This ensures that

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

[7]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 types of

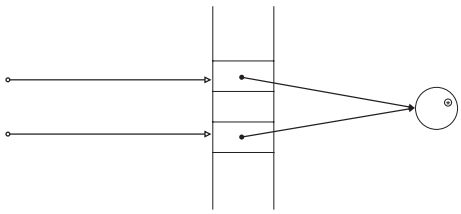


Figure 6. Multiple Naming: Two entries in the object table can hold the same memory address. Comparison must yield true, because it is based on the single comparand stored in the object.

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

3.2: Multiple Naming

The reference assignment operator #= introduces the possibility that different references may refer to the same object, a case that cannot occur in a pure Java program, but only when this operator is applied. What happens here is that different object table entries may hold the same memory address, as is depicted in fig. 6.

This also illustrates one of the reasons why our model does not allow references to be compared — it is not clear whether an attempt to compare references should actually be performed on the OOPs or on the memory addresses. Comparison of the OOPs in fig. 6 would indicate the difference of the references whereas comparison of the memory addresses would indicate sameness. For this reason, other proposals call for means to merge objects and their identities into a single object in order to avoid the described ambiguity of reference comparison. For example, [4] proposes an operation MakeSameObject(...) that has the described effect.

However, the combination of different OOPs into one entry after application of the reference assignment operator #= would actually lead to unexpected results. For example, the merge of the two references "The President of the USA" and "Bill Clinton" would either mean that a change of the former would also change the latter, or otherwise a change of any of the involved references would have to be prevented altogether. Both options are clearly unacceptable.

In our model, OOPs are never compared, but comparison is performed on the comparands stored in the involved objects. Since comparands are properties of their respective objects, this should also be easier to remember by a programmer than an arbitrary decision to base comparison operations on either OOPs or memory addresses.

Since the two references in fig. 6 refer to the same object, in this case the comparison operation does not have a choice and must unambiguously indicate sameness. If one wants to distinguish between the two references, one of them can be redirected to a different object that forwards all messages to the original object. For example, the reference "The President of the USA" may refer to a kind of "role" object decorating the object referred to by "Bill Clinton" by appropriately applying the reference assignment operator #= (see

exception can be avoided by testing variables against null beforehand.

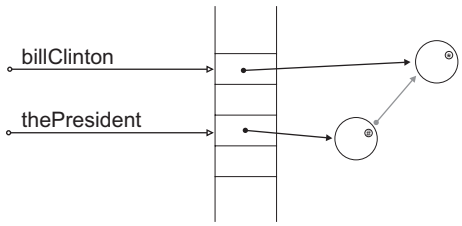


Figure 7. Multiple Naming: Insertion of a decorator object enables one to distinguish between two references. (Contrary to what the simpli£ed illustration suggests, the grey arrow is actually a pointer into the object table, not a "direct" reference to the target object.)

fig. 7). This also allows a programmer to define methods differently within such a "role" or decorator object so that the behaviour of "Bill Clinton" can vary depending on whether he is approached as "The President of the USA" or privately. The result of the comparison operation may be further influenced by manipulating the comparands in the involved objects, as shown in the following paragraphs.

3.3: Operations on Comparands

It is obvious from a technical point of view that comparands may be freely copied between objects. There are, in fact, good reasons on a conceptual level to allow copying comparands. When a programmer introduces decorator objects, they usually have to "take over" the comparand of the wrapped object so that comparison operations that involve "direct" references to a wrapped object yield the correct result.

Comparands are introduced in GILGUL by defining a pseudo-class Comparand in the package java.lang.*. This class can only be used to create new comparands via class instance ("comparand") creation expressions (new Comparand()), but not as a reference type in the declaration of variables. The only place where it can be understood as being used as a type is in the definition of java.lang.Object, 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().

Since comparands cannot directly be manipulated, there are no limitations on how they are implemented in a concrete virtual machine. The only requirements they have to fulfil are as follows.

- 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).

The comparand of the null literal null 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 non-ambiguous.

3.4: Operations on Objects

Besides the operations on references and comparands that are newly introduced in GILGUL, the operations on objects, i.e. the methods defined in their corresponding classes, are still available as a matter of course. However, there are some interdependencies between certain standard methods, namely equals(...) and hashCode(), and the ability to copy comparands between objects.

Note that the standard definition of equals(...) relies on the definition of the equality operators == and !=, and therefore is affected by the fact that they have been redefined to operate on comparands instead of references. Hence, it yields true when the comparands of the corresponding objects are the same. As a consequence, the standard definition of hashCode() has been changed to return a hash code value for an object's comparand, since the contract of hashCode() is based on equals(...).[8]

From this perspective comparands can be seen as an additional way to redefine the method equals(...) by just copying them between objects. In certain cases, this may suit the programmer's imagination better than having to override equals(...) and for example send appropriate comparison messages to referenced objects. Furthermore, they relieve the programmer of the requirement to remember to override hashCode() accordingly whenever he or she is about to redefine equals(...).

Another consequence is that the equality operators == and != and the method equals(...) are always redefined in a uniform way by copying of comparands, unless equals(...) is explicitly overridden by the programmer. This complies with the suggestion that there should be only one comparison operation per object, as is stated for example in [2][9] or [6][10]. From this perspective, the decision to include two different ways to compare objects into the Java programming language may be considered questionable.

3.5: Application to Dynamic Tailoring

Returning to our given problem, we are now able to apply the new operations to achieve the desired insertion of a security component atomically. We could apply stubObject #=

[8] [13] states that if "two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result." A comparand's hash code value fulfils the same requirement, except that it is based on the equality operator on comparands (==) instead of the method equals(...) which is not supported by comparands.

[9] It states that a programming language should provide "only one copy method and one comparison method for each class. The designer of the class, rather than its clients, should choose appropriate semantics for these methods."

[10] "Use method equals instead of operator == when comparing objects. [...] Rationale: If someone defined an equals method to compare objects, then they want you to use it. Otherwise, the default implementation of Object.equals is just to use ==."

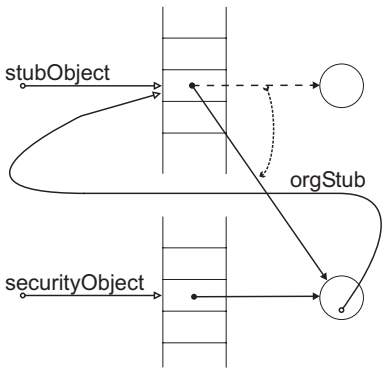


Figure 8. Naive application of stubObject #= securityObject **results in an unwanted cycle: When** securityObject.orgStub **holds the same reference as** stubObject **beforehand, it will refer to** securityObject **afterwards.**

securityObject to let securityObject "take over" the identity of stubObject. However, one has to be careful, because securityObject certainly has to refer to the original stubObject in order to delegate messages that it cannot handle for itself. Regard the following naive sequence of operations.

```
securityObject.orgStub = stubObject;
stubObject #= securityObject;
```

This would be erroneous, because afterwards securityObject.orgStub would refer to securityObject since it contains the same reference as stubObject according to the first simple assignment. This of course results in a cycle, and therefore to non-terminating loops for messages that cannot be handled by securityObject (see fig. 8). The following sequence however is correct (see fig. 9).

```
tmp #= stubObject;            //let a new reference point to stubObject

securityObject.orgStub = tmp; //tmp instead of stubObject

securityObject.comparand
      = stubObject.comparand; //ensure that equality behaves well

stubObject #= securityObject; //tmp and so securityObject.orgStub remain unchanged
```

Note that the actual "replacement" of stubObject is initiated by the last operation, and thus is indeed atomic.

As we can see the operations that are newly introduced in GILGUL give the programmer the possibility to "replace" the former stub object atomically and thereby introduce the security feature without having to deal with any consistency problems. Furthermore, the involved components need not anticipate such modifications, reducing the complexity of the development of the actual components to a great extent.

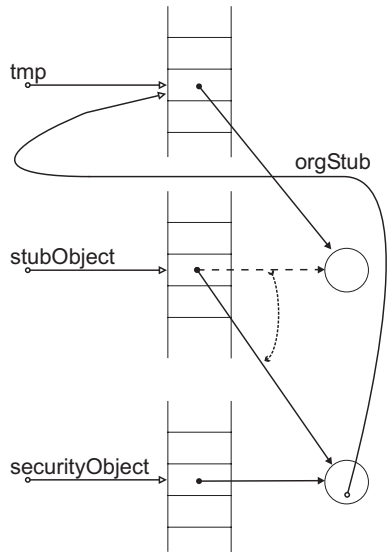


Figure 9. Correct application of stubObject #= securityObject**: When** securityObject.orgStub **holds a different reference to the same object as** stubObject **beforehand, it will still refer to the former** stubObject **afterwards, since the temporary reference is not affected by this operation. Therefore, the unwanted cycle is avoided.**

4: Related Work

There are only a few works discussing the concept of object identity and the incorporated notions of reference and comparison, for example [7] which appears to be the first work that gives a detailed differentiation of values and objects, and [5] which focuses the discussion on the properties of object identity rather than the differences in values. In both works the notions of reference and comparison are implicitly subsumed in the concept of object identity.

[14] gives a formal model of object identity that again incorporates both notions of reference and comparison. It introduces a set of requirements with the purpose of ensuring that "in an administration that uses a single oid scheme, we always can count objects by counting oids." The act of counting objects is closely related to the notion of comparison, so the emphasis of the discussion in that paper lies on this notion. Our work actually shows that these requirements can be abandoned if we clearly separate the notions of reference and comparison.

[4] also discusses various aspects of object identity. It argues that object identity is not about comparing objects but about referencing them. "Deciding which things are the same is very carefully excluded from the model." It introduces the concept of synonymous handles, that are essentially different identifiers referencing the same object. For example, an operation MakeSameObject(...) is sketched. This is similar to the case of multiple naming in our model, however we have shown that there is no need to actually merge the OOPs involved because they are never compared.

[2] gives a thorough examination of several variations of copying and comparison operations. For example, it shows that there are three copying operations, namely "assign", "replace" and "clone". The comparison operations are divided into variations of identity,

shallow and deep equality, stressing the problematic subtleties, especially of the last two that are generally considered reasonable and harmless. Put very roughly, both copying and comparison operations can be separated into identity-based and value-based. Our approach focuses on the identity-based operations and explores new possibilities in this realm, intentionally ignoring the value-based approaches. In this sense, that paper is orthogonal to our work.

The programming language Smalltalk provides an operation become: that enables one to "swap" the objects pointed to by two references without actually changing the references. Since most Smalltalk implementations are based on object tables, this can very easily be realised. This is similar to what can be expressed by our reference assignment operator #=. For example, see the Smalltalk FAQ [8] for further details.

5: Summary

In this paper we have demonstrated severe consistency problems stemming from the restrictions imposed on the concept of object identity in object-oriented programming languages. We have shown this in the context of a concrete component-oriented software system that allows the reconfiguration of components during runtime.

We have then given a detailed differentiation between reference and comparison, two notions that are usually subsumed in the concept of object identity. Most probably the main reason for this subsumption lies in the fact that physical memory addresses can be used to implement both purposes in an extremely efficient way. However, if we separate the two notions, we can greatly increase the expressiveness of a programming language.

We have designed the programming language Gilgul, a compatible extension to Java. It introduces very few constructs: the pseudo-class Comparand and the reference assignment operator #=. It also changes the definition of the existing equality operators == and != according to our model.[11]

Our model is a generalization of what can be expressed in terms of object identity in current object-oriented programming languages. Our model also greatly simplifies the redefinition of the standard method equals(...), a task that is needlessly cumbersome and error-prone in pure Java.

We have shown how the operations that are newly introduced in Gilgul can be applied on an example of dynamic recomposition. They allow the programmer to achieve the desired replacement of components atomically, thereby completely avoiding the mentioned consistency problems.

However, there are some issues that we have not dealt with in this paper. We have investigated the problem of ensuring type soundness when applying the reference assignment operator #=. In essence it must be ensured that the set of types of an object a reference is redirected to by applying this operator has to be equal to or a superset of the types of the former object. In order not to reduce the set of assignable objects too much by this requirement we have introduced new (light-weight) constructs into Gilgul's type system. There are also some semantic safety issues that are not related to the type system. The results of these investigations will be reported elsewhere.

[11] A compiler and runtime system for Gilgul is currently being developed at the Institute of Computer Science III of the University of Bonn.

As a bottom line, GILGUL is not a totally new approach of dealing with issues of object identity. As can be seen from the related work, many of its ideas have come across previously. Nonetheless, GILGUL is the first approach known to the authors that strictly and cleanly separates the notions of reference and comparison on the level of a programming language, and in this way throws new light on the concept of object identity.

6: Acknowledgements

The authors would like to thank their colleagues at the Institute of Computer Science III of the University of Bonn for many fruitful discussions. They also would like to particularly thank Tom Arbuckle, Michael Austermann, Peter Grogono, Arno Haase, Günter Kniesel, Thomas Kühne and Kris De Volder for their critical review of earlier drafts of this publication which led to substantial improvements. This work is partially located in the TAILOR project at the Institute of Computer Science III of the University of Bonn and has been 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] P. Grogono and M. Sakkinen, *Copying and Comparing: Problems and Solutions*, in: E. Bertino (ed.), *ECOOP 2000 — Object-Oriented Programming*, Proceedings, Springer LNCS 1850, 226-250, June, 2000.

[3] JavaSoft, *JavaBeans 1.0 API Specification*, Version 1.00-A ed. Mountain View, California: Sun Microsystems, 1997.

[4] W. Kent, *A Rigorous Model of Object References, Identity and Existence*, Journal of Object-Oriented Programming, 4(3):28-36, June, 1991.

[5] S. N. Khoshafian and G. P. Copeland, *Object Identity*, OOPSLA '86 Proceedings, 406-416, September, 1986.

[6] D. Lea, *Draft Java Coding Standard*, `http://gee.cs.oswego.edu/dl/html/javaCodingStd.html`, February, 2000.

[7] B. J. MacLennan, *Values and Objects in Programming Languages*, SIGPLAN Notices, 17(12):70-79, December, 1982.

[8] D. N. Smith, *Dave's Smalltalk FAQ*, `http://www.dnsmith.com/SmallFAQ/`, July, 1995.

[9] O. Stiemerling, *Component-Based Tailorability*, Ph. D. Thesis, in Institute of Computer Science III, University of Bonn, 2000.

[10] O. Stiemerling, R. Hinken, and A. B. Cremers, *Distributed Component-based Tailorability of CSCW Applications*, in: Proceedings of ISADS '99, Tokyo, Japan, IEEE Press, pp. 345-352, 1999.

[11] O. Stiemerling, R. Hinken, and A. B. Cremers, *The Evolve Tailoring Platform: Supporting the Evolution of Component-Based Groupware*, in: Proceedings of EDOC '99 (Enterprise Distributed Object Computing), Mannheim, Germany, IEEE Press, pp. 106-115, 1999.

[12] O. Stiemerling, *FlexiBeans Specification V 2.0*, Institute of Computer Science III, University of Bonn, Working Paper, 1998, (`http://www.cs.uni-bonn.de/~os/Publications/Flexibeansv20.ps`).

[13] Sun Microsystems, Inc., *Java 2 SDK, Standard Edition Documentation, Version 1.3*, `http://java.sun.com/products/jdk/1.3/docs/`, 2000.

[14] R. Wieringa and W. de Jonge, *Object Identifiers, Keys, and Surrogates — Object Identifiers Revisited*, Theory and Practice of Object Systems, 1(2):101-114, 1995.