

Object-Oriented Composition Untangled

Klaus Ostermann
Siemens AG, Corporate Technology SE 2
D-81730 Munich, Germany
Klaus.Ostermann@mchp.siemens.de

Mira Mezini
Darmstadt University of Technology
D-64283 Darmstadt, Germany
mezini@informatik.tu-darmstadt.de

ABSTRACT

Object-oriented languages come with pre-defined composition mechanisms, such as inheritance, object composition, or delegation, each characterized by a certain set of composition properties, which do not themselves individually exist as abstractions at the language level. However, often non-standard composition semantics is needed, with a mixture of composition properties, which is not provided as such by any of the standard composition mechanisms. Such non-standard semantics are simulated by complicated architectures that are sensitive to requirement changes and cannot easily be adapted without invalidating existing clients. In this paper, we propose *compound references*, a new abstraction for object references, that allows us to provide explicit linguistic means for expressing and combining individual composition properties on-demand. The model is statically typed and allows the programmer to express a seamless spectrum of composition semantics in the interval between object composition and inheritance. The resulting programs are better understandable, due to explicitly expressed design decisions, and less sensitive to requirement changes.

1. INTRODUCTION

The two basic composition mechanisms of object-oriented languages, inheritance and object composition, are very different concepts, each characterized by a different set of properties. The properties of inheritance have been discussed in several works, e.g., [25, 31, 22]. Also, the relationship between inheritance and object composition is carefully studied, e.g., in [17, 16]. The mixture of composition properties supported by each mechanism is fixed in the language implementation and individual properties do not exist as abstractions at the language level.

However, often non-standard composition semantics is needed, with a mixture of properties, which is not as such

provided by any of the standard techniques. We indicate that in the absence of linguistic means for expressing and combining individual composition properties on-demand, such non-standard semantics are simulated by complicated architectures that are sensitive to requirement changes and cannot easily be adapted without invalidating existing clients. Actually, the need to combine properties of inheritance and object composition has already been the driving force for two families of non-standard approaches to object-oriented composition.

On one side, *delegation* [20] enriches object composition with inheritance properties. Please note that in contrast to the frequent use of the term *delegation* as a synonym for forwarding semantics, in this paper it stands for dynamic, object-based inheritance. In pure delegation-based models, objects are created by cloning other prototype objects, and objects may inherit from other objects, called *parents*. Hence, in such models one has object composition and delegation, but no class-based inheritance. The most prominent programming language in this family is SELF [32]. More recently delegation-based techniques are integrated into statically typed, class-based languages, which thus provide class-based inheritance, delegation, and object composition [18, 11, 4]. On the other side, several *mixin*-based models [7, 21, 12, 2] approach the goal of combining inheritance and object composition properties from the opposite direction, enriching inheritance with object composition properties, such as the ability to statically/dynamically apply a subclass to several base classes.

Like standard composition mechanisms, these approaches also do not provide abstractions for explicitly expressing individual composition properties that would allow to combine these properties on-demand. In this paper, we distinguish between five properties that can be used to describe the relation that holds between two modules M and B (classes and/or objects) to be composed, whereby B denotes the base module, M denotes the modification module, and $M(B)$ denotes the composition.

1. **Overriding:** The ability of the modification to override methods defined in the base. In $M(B)$, M 's definitions hide B 's definitions with the same name. Self-invocations within B ignore redefinitions in M .
2. **Transparent redirection:** The ability to transpar-

To appear in proceedings of OOPSLA 2001. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

	inheritance	object composition	delegation	mixin inheritance
overriding	x	-	x	x
redirection	x	-	x	x
acquisition	x	-	x	x
subtyping	x	-	x	x
polymorphic	-	dynamically	dynamically	statically

Table 1: Composition properties supported by standard mechanisms

ently redirect B’s `this` to denote $M(B)$ within the composition.

- Acquisition:** The ability to use definitions in B as if these were local methods in $M(B)$ (transparent forwarding of services from M to B).
- Subtyping:** The promise that $M(B)$ fulfills the contract specified by B , or that $M(B)$ can be used everywhere B is expected.
- Polymorphism:** The ability to (dynamically or statically) apply M to any subtype of B .

Table 1 shows the set of properties we discuss in the paper as row indexes. Columns are indexed by existing object-oriented composition mechanisms.

The key idea of the approach presented in this paper is the *separation* and *independent applicability* of these notions by providing explicit linguistic means to express them. This allows the programmer to build a seamless spectrum of composition semantics in the interval between object composition and inheritance, depending on the requirements at hand, making object-oriented programs more understandable, due to explicitly expressed design decisions, and less sensitive to requirement changes, due to the seamless transition from one composition semantics to another.

The remainder of the paper is organized as follows. Sec. 2 discusses examples where non-standard combinations of composition properties are desirable. Sec. 3 presents the basic concepts of our model. The model is evaluated in Sec. 4. Sec. 5 discusses some advanced issues related to static type safety. Related work is discussed in Sec. 6. Sec. 7 summarizes the paper and suggests areas of future work.

2. MOTIVATION

In this section we consider three composition scenarios where non-standard combinations of composition properties make sense. In all cases, we discuss various designs that can be used to achieve the desired composition semantics. However, please note that this section *is not about* proposing THE ultimate designs for the given scenarios. The reader might eventually come up with other, equivalent or even superior, designs to the same scenarios. Yet, this is not essential for the purpose of this chapter: The main message

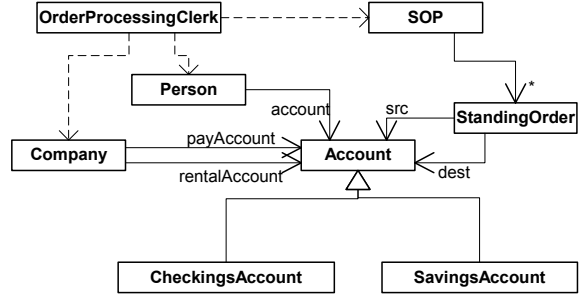


Figure 1: Class diagram for account example

we want to convey is rather (a) that in all cases *some* sophisticated design is needed, which does not explicitly state important conceptual relationships between the involved abstractions, and (b) that different designs are needed for different combinations of composition properties.

2.1 Composition Scenario 1: The Account Example

Consider an application in the banking domain with persons, companies, accounts, and standing orders. The relation between persons/companies and accounts is usually one to many. However, in this example we want each account to have a dedicated role for its owner. For example, we want a company to have a dedicated pay account and a dedicated rental account. This makes it possible to choose the appropriate account for specific transfers automatically. In this (simplified) example, a `Person` has only one “main” account and a `Company` has a rental and a pay account. Different kinds of accounts exist (`SavingsAccount`, `CheckingsAccount`), and accounts are subject to frequent changes at runtime. A particular account may be shared, as e.g., two persons may use the same account, or the pay account and the rental account of a company may be identical. A class `SOP` (standing order processing) is used for the registration, deregistration and execution of standing orders. On execution, multiple standing orders with identical source and target accounts are summarized to a single transfer.

A class diagram for this problem is shown in Fig. 1. Based on the information on a pay order, the `OrderProcessingClerk` gets the account objects from the involved `Person/Company`, creates a `StandingOrder` and registers it with a `SOP`. This design is simple and easy to understand. However, it has a problem: If the account of a person changes, a previously registered standing order will still be executed with respect to the outdated account. With the design in Fig. 1, one has to update all account references that were ever given out by a person or company “manually”. That is definitely undesirable.

Given a reference p to a `Person` object, we ideally want “ p ’s account”, i.e., a compound, or indirect reference to `account` via p , to be passed to `SOP`, rather than the `account` reference itself. In other words, we want some kind of *redirect semantics* for references: the meaning of `account` should be

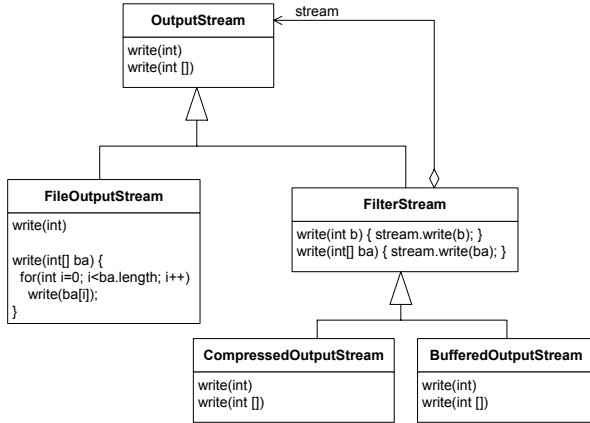


Figure 2: Decorator design for output streams

late bound within the current context of the object referred to by *p*, whenever “*p*’s account” gets evaluated. Due to the lack of such “compound references” in standard object-oriented languages, we have to change the architecture of our design to simulate them. Some possible solutions are discussed below.

- A *decorator* [15] that contains an account object and forwards all calls to it is passed to *SOP* instead of the account object itself. The base object of the decorator (the account) can be changed without the need for further manual updates. However, the identity test in *SOP* fails: If two persons share an account, *SOP* compares non-identical decorators with the same base object. Other subtleties of an architecture that uses the decorator pattern for composition are highlighted in the next scenario.
- A second approach is to change the *SOP* class so that it accepts *AccountOwner* instead of *Account* objects with *AccountOwner* being an interface with a single *getAccount()* method. *Person* and *Company* have to implement the *AccountOwner* interface. This is difficult in the *Company* case, because a company has two different accounts. For this reason, we have to create a separate *AccountOwner* subclass for *Company*¹. Besides its complexity, the main drawback of this approach is that we have to modify the *StandingOrderProcessing* class, which might not be desirable or even possible in case this class is purchased as part of a banking component library. Another limitation of this proposal is that it works only for one level of redirection. For example, we might want to register a standing order that transfers money to the account of a person’s current spouse (and, spouses are also subject to frequent changes these days).

- A common approach to avoid the coupling of the

¹In Java, these classes would probably be implemented as inner classes.

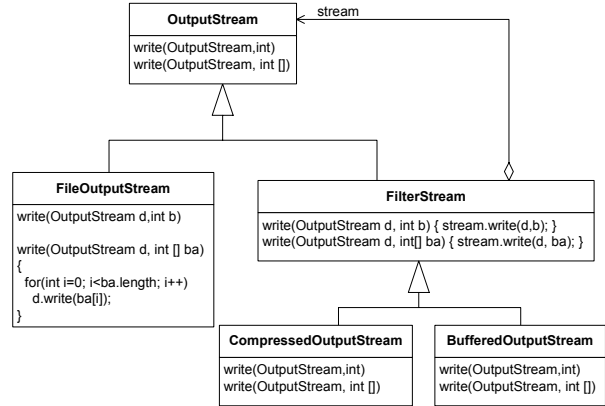


Figure 3: Simulation of transparent redirection

sender of a request to its receiver is the chain of responsibility pattern [15]. Applied to our example this would mean that each account has an optional successor account and new accounts are appended to the previous account. Calls to the account are forwarded to the last and current account in the chain. Besides its considerable complexity, this approach is not compatible with sharing of accounts.

- Another possible solution would be to let the *SOP* class be an observer [15] of persons and companies that is notified whenever an account is exchanged. However, it is easy to see that this would result in a design that is even more complicated than the previous ones.

2.2 Composition Scenario 2: The Stream Example

I/O Streams exist in multiple variations and the different stream features are typically implemented as decorators [15] of a basic stream class (see e.g., the Java I/O package [29]), so that the set of desired features for a stream instance can be chosen dynamically. A typical decorator design for output streams is shown in Fig. 2.

By using the decorator to compose basic *OutputStream* functionality with optional filtering features we want to achieve the following composition properties: (a) subtyping between the resulting composition and the base component, (b) acquisition of the base behavior within the filtering functionality, (c) dynamic polymorphism - a certain filtering should be applicable to any subtype of *OutputStream*, and (d) overriding of the methods that have to be changed for the extended functionality. The decorator pattern realizes dynamic polymorphism of the composition by means of object composition and subtype polymorphism. Acquisition and overriding is achieved by implementing the base component’s interface by means of forwarding methods resp. decorator-specific methods. The decorator becomes a subtype of the component by inheritance. The simulation of these composition properties, however, has a number of shortcomings:

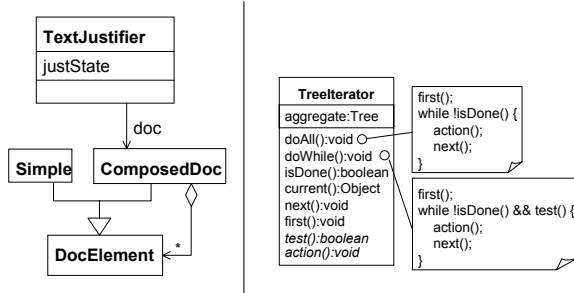


Figure 4: Structure of text justifier and tree iterator

- The implementation of the decorator class is a tedious and error-prone work, due to the manual simulation of the acquisition feature. In addition, it suffers from the *syntactic fragile base class problem* [30]: Whenever the interface of the base component changes, the corresponding forwarding methods have to be added or deleted.
- We have no transparent redirection. This means that method calls to `this` within the base component are not dispatched to their overridden methods in the decorators but to the local implementations. A further consequence is that if a base object passes `this` to other objects, it passes itself instead of the decorator. In some situations, however, the opposite effect would be desirable. This anomaly is known as the *self problem* [20] or as *broken delegation* [16]. The manual simulation of transparent redirection is rather complex and leads to a design that is very different from the original decorator pattern because the base object needs some way of knowing about the decorators. One alternative is to store a back reference to the decorator in the base, but this would prohibit multiple decorators for one base object. Another solution with (again) a different design is to pass the decorator to the base object in every method call. A corresponding design is illustrated in Fig. 3.
- The base class may define state. All decorators inherit this state and become unnecessary heavy. Although merely a subtype relationship between the decorator and the base component is needed, the decorator is enforced to inherit the state, due to the use of inheritance for subtyping. This is usually no problem if the usage as a base for decorators was already anticipated at writing time. However, if a predefined library class should be decorated, this may be a problem.

2.3 Composition Scenario 3: The TextJustifier Example

Envisage a `TextJustifier` command class in a text processing system, which justifies all paragraphs in a document, except for preformatted paragraphs. The document elements to be justified are stored in a recursive object structure, as shown in the diagram on the left-hand side of Fig. 4².

²In a more realistic situation, one would have to apply the visitor pattern to connect `TextJustifier` and the `DocElement`

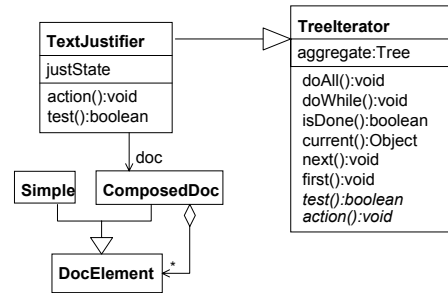


Figure 5: Usage of the iterator by inheritance

For performing the document justification the text justifier needs to iterate over the document structure. Assume that we have already implemented a tree iterator class shown on the right-hand side of Fig. 4. The class `TreeIterator` encodes a breadth-first iteration strategy for recursive object structures. It can be used by overriding the `action()` and `test()` methods for the specific purpose. The iterator class provides a number of iteration mechanisms, e.g., applying `action()` to all elements that satisfy `test()` (`doAll()`), or up to the first one that does not satisfy `test()` (`doWhile()`), and so on. Assume that the design shown in Fig. 5³ where text justification and iteration functionality are composed by means of inheritance is just good enough for satisfying the requirements on our system during an early stage of the development process.

In a later iteration stage, we realize that inheritance is not the composition semantics we want. First, we do not want `TextJustifier` to be a subtype of `TreeIterator` anymore because a `TextJustifier` is not a special kind of an iterator. In addition, the acquisition semantics that comes with inheritance is not desired anymore; all methods of `TreeIterator` pollute the interface of `TextJustifier`, which has become complex anyway during the development. Second, the initial requirements have slightly changed: It should be possible to determine the iteration strategy to be used with a `TextJustifier` at runtime. For this purpose, subclasses `PreOrder` and `PostOrder` of `TreeIterator` have been implemented that refine the default breadth-first semantics by overriding the `first()` and `next()` methods.

Now, the question is how to compose the text justifier in Fig. 4 with the iteration hierarchy, such that the above set of composition properties are satisfied. A feasible solution is schematically presented in Fig. 6. `TextJustifier` has an instance variable, `it`, of type `TreeIterator`, which can be assigned to an instance of `MyPreOrder`, `MyIterator`, or `MyPostOrder`. The latter are defined as subclasses of the corresponding library classes and redundantly implement the `test()` and `action()` methods for the justification purposes. It is quite reasonable to assume that the test and the action performed in each step of the iteration needs infor-

ment hierarchy. For the sake of simplicity, we assume this is not the case in our example. The problems we discuss here apply to a visitor-based design as well.

³In the design we assume that `DocElement` implements `Tree`

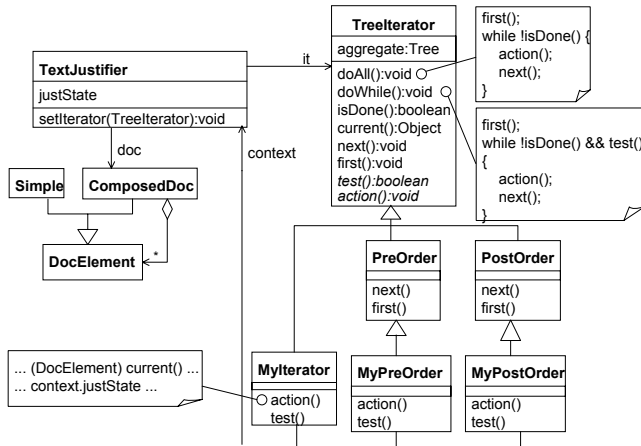


Figure 6: Initial design for dynamic composition

mation from the text justifier object, which is provided via the `context` reference on the `TextJustifier`.

Obviously, the design in Fig. 6 is very different from the predecessor design in Fig. 5. That is, two different mixtures of features for composing the same pieces of functionality are realized by two very different designs. Furthermore, the design is more complex than the design in Fig. 5, and it does not reflect the conceptual relationships between the entities in it. Additional classes and associations have been introduced, and the `MyXXX` classes contain duplicated implementations of `action()` and `test()`.

At this point, it becomes clear that the initial iterator design is unsatisfactory because it leads to code duplication as in Fig. 6. It would have been better to choose a more sophisticated design for the iterator classes right from start, namely iterators that use a command class (`IterationStep`) as shown in Fig. 7. But this still does not solve our problem: We still need a complex design such as in Fig. 7, although the conceptual relationship between `TextJustifier` and `TreeIterator` is as simple as the initial design in Fig 5. The design in Fig 5 would already be sufficient, even for the dynamic composition, if only we could configure the relation between `TextJustifier` and `TreeIterator` with the properties in Tab. 1.

2.4 Problem Statement Summary

So far so good. In all cases, the desired composition semantics can indeed be achieved somehow. Still, the result is highly unsatisfactory. Why? First, the most severe problem is that the architectures we ended up with are completely different, depending on the desired mixture of composition properties. Second, the design gets complex, as soon as a composition is required that deviates from the semantics of the standard composition mechanisms directly supported by linguistic means.

As illustrated by the first composition scenario, different programmers may come up with different architectures even

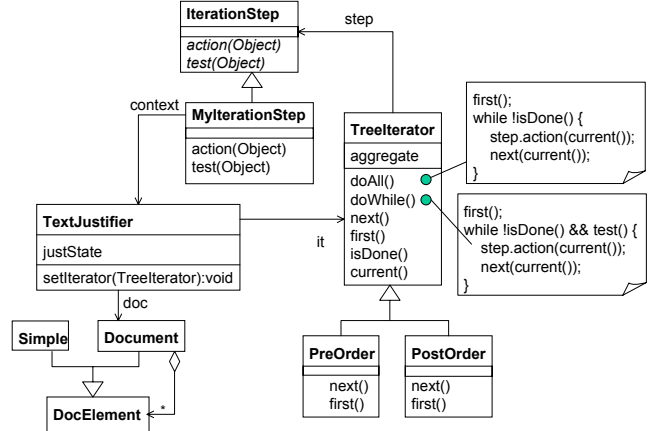


Figure 7: More sophisticated design for dynamic composition

for the same composition semantics. Moreover, any later change of the composition features might require switching to another architecture. This is not only a tough challenge for any programmer. It also affects the understandability, and hence maintenance of object-oriented programs: The very important knowledge about the encoded composition semantics is not explicitly expressed by any of the designs that simulate non-standard semantics. In general, it is not obvious how to separate the part of the architecture that is directly involved in encoding application logic from the part of the architecture that serves as an infrastructure for encoding non-standard composition semantics. As a consequence, it is hard to guess from looking at the design that two architectures are different merely because they encode different composition semantics, or that two different architectures actually implement the same application logic, and only differ in the way they encode the same composition semantics.

The frequency of changes in the composition features is documented by *refactorings*, such as, “Replace Delegation with Inheritance”⁴, “Replace Inheritance with Delegation”, “Hide Delegate” and “Remove Middle Man” [13]. In our terminology, each of these refactorings can be seen as moving from an architecture with a certain mixture of composition properties to another one with another mixture of composition features that better fits the requirements or the current state of the development process. The work on refactoring recognizes that such transformations are not trivial and aims at aiding programmers in performing them by describing the process in a systematic way, or even (partly) automating it by means of refactoring browsers. The highly positive echo that the work on refactoring has found in the object-oriented community, especially in the practice of everyday programming, actually supports our claim that the need for different architectures to express different composition semantics makes a programmer’s life harder.

⁴Fowler uses the term delegation in the sense of decorator-like forwarding

However, we follow another path in approaching the problem, motivated by the observation that identifying and describing common refactorings does not solve the core problem: It does not change anything in the fact that different architectures for different composition semantics are needed and one still needs to switch from one architecture to the other in order to react to requirement changes. Consequently, we put the emphasis on tackling the problem in its roots: At language design. Our claim is that besides identifying and describing refactorings we should strive for language mechanisms that make some of the refactorings obsolete, or at least explicit in the language. This requirement becomes even more relevant in a component setting where refactoring steps like “adjust all clients to call the new server” are no longer feasible.

3. THE COMPOUND REFERENCE MODEL

This section introduces the basic notions of our model as an extension of the Java programming language [3]. However, the concepts are easily applicable to other statically typed OO languages. Each introduced feature corresponds to a row in Table 1 and represents a step forward on a seamless transition from object to inheritance-based composition semantics.

3.1 Field Methods and Overriding

To explain the operational semantics of our model, we use the notion of *field methods*. A *field method* is a method that pertains to a specific field. Syntactically, the affiliation of a method to a field is expressed by prefixing the method name with the field name using “.” as separator. A class *C* with a field *f* of type *F* can be thought of as implicitly containing a method named *f.m()* for every public method *m()* of *F*. The method named *f.m()* has the same signature as *m()* in *F*, and its visibility is identical to the visibility of *f* in *C*. The default implementation of *f.m()* in *C* is one that simply forwards *m()* to the object referred to by *f*, denoted within the implementation of *f.m()* by the special pseudo-variable *field*. This is similar to the pseudo-variable *super* denoting an overridden method within the overriding method. Finally, any invocation of *m()* on the object referred to by *f* within *C* should be thought of as being dispatched to the corresponding implicit field method *f.m()*.

For illustration, recall the iterator example from Sec. 2. With the implicit field methods written down, the code for the `TextJustifier` would look like in Fig. 8⁵. The call `it.doAll()` within `justify()` should be thought of as actually calling the implicitly available field method named `it.doAll()`.

Until now, the introduction of field methods into the implementation of a class has no impact on the semantics of the class. The `TextJustifier` implementation presented in Fig. 8 is semantically equivalent to an implementation that does not contain any implicit field method. The decisive point is that implicit methods can be replaced by explicitly

⁵In the context of this section, the reader should think of the abstract methods as being implemented empty.

```
class TextJustifier {
    private TreeIterator it;
    public void justify() { ... it.doAll() ... }

    // ** begin of implicitly available field methods **

    private void it.doAll()      { field.doAll(); }
    private void it.doWhile()   { field.doWhile(); }
    private void it.doUntil()   { field.doUntil(); }
    ...
    private void it.action(Item x) { field.action(x); }
    private boolean it.test(Item x) {
        return field.test(x);
    }
    // ** end of implicitly available field methods **
}
```

Figure 8: Implicit field methods in `TextJustifier`

```
class TextJustifier {
    private TreeIterator it;

    private void it.doAll() { ... }

    public void justify() {... it.doAll(...); }
}
```

Figure 9: Explicit field methods in `TextJustifier`

available methods. For example, in order to implement an action to be undertaken whenever `it.doAll()` is called in `TextJustifier`, the programmer of `TextJustifier` would implement an explicit field method, called `it.doAll()`, encoding the desired behavior, as shown in Fig. 9. Please note that Fig. 9 is only an illustration of explicit field methods and not our final solution for the `TextJustifier` problem.

Before leaving this section we would like the reader to recall that we introduced implicit methods as a means to describe the operational semantics of our model, independently of a specific implementation.

3.2 Field Redirection with Compound References

The central mechanism of our model is the notion of *compound references* (CR). In contrast to “primitive references”, the binding of a CR to an object is not absolute, but rather relative to another reference. To gain a first insight for the usefulness of CRs reconsider the account example from Sec. 2.1. We could solve the problem discussed there if we were able to express that “*person’s account*” - meaning the *account* reference within the context of the *person* reference - should be passed to the standing order processing unit. This is where CRs come into play.

A CR to a reference `instVar` within a class is created by means of `this<-instVar`. To illustrate their semantics, consider the class in Fig. 10⁶. The `getPersonsAccount()` method returns a compound reference to the *account* in-

⁶Please note that in Java all object-typed instance variables are references.

```

class Person {
    Account account;
    Account getAccount() { return account;}
    Account getPersonsAccount() {
        return this<-account;
    }
    void setAccount(Account newAccount) {
        account = newAccount;
    }
}

class Client {
    public static void main(String[] args) {
        Person jack = ...
        Account ubsAccount = new Account("UBS", "12345");
        Account dbAccount =
            new Account("Deutsche Bank", "54321");
        jack.setAccount(ubsAccount);
        Account anAccount = jack.getAccount();
        Account jacksAccount = jack.getPersonsAccount();

        // anAccount and jacksAccount
        // refer to the UBS account

        jack.setAccount(dbAccount);

        // anAccount still refers to the UBS account
        // but jacksAccount refers to the DB account
    }
}

```

Figure 10: Illustration of compound references

stance variable of a person, while `getAccount()` returns a “primitive” reference to the `account` instance variable of a person. The effect of the CR returned by `getPersonsAccount()` is that it always refers to the current value of the `account` reference within a `Person` object. After the `setAccount` call in the last statement of `Client::main` in Fig. 10, which changes Jack’s account from `ubsAccount` to `dbAccount`, `jacksAccount` will refer to Jack’s current “Deutsche Bank” account, while `anAccount` will still refer to his old UBS account. Fig. 11 and Fig. 12 schematically show the state before and after changing Jack’s account.

Just like object methods that differ from functions in the sense that different calls to them may return different values, depending on the state of the method’s owner (the receiver), a CR is different from a primitive reference in the sense that the evaluation of a CR might result in different values depending on the state of CR’s owner object. CRs are very different from pointers or pointers to pointers etc. A pointer always explicitly specifies the dimension of indirection (in C++ the number of `*`). Pointer of different dimensions (for example, `Account **a1` and `Account ***a2`) are not compatible or substitutable⁷. CRs, on the other hand, are a transparent replacement for usual references: It is generally not known whether a reference is a CR or not, or how many levels of indirection are hidden in the CR. In a way, CRs are similar to symbolic links in a Unix file system.

⁷A *conversion* from `**` to `***` is possible in C++, for example `a1 = *a2`, but the semantics is different: If the first indirection of `a2` is changed after this assignment, `a1` still points to the previous account.

A symbolic link may refer to a file or to another symbolic link. If objects were directories, we could create a symbolic account link in the `SOP` directory that refers to the account link in the `Jack` directory.

A CR can be defined relatively to a primitive reference or recursively to another CR. Hence, each CR may in general induce a *path* of object references. For example, a class `Person` might return a CR to the spouse of that person. If the `getPersonsAccount()` method on this CR is called, we obtain a new CR with path `personOID<-spouse<-account`. In general, a CR is an OID `o` together with a sequence of field names v_1, \dots, v_n . A CR `o<-v1<-...<-vn` induces a corresponding path of objects $o_0<-o_1<-...<-o_n$ such that $o_0 = o$ and $o_i = o_{i-1}.v_i$ (details and subtleties about creating an object path for a CR are discussed in section 5). Such a path is not created directly but incrementally as a result of creating a CR to a reference that is actually already a CR. In the following, we regard a “usual” reference as a special case of a CR of length one. Relative to an element o_i , we call o_{i-1} a *predecessor* and o_{i+1} a *successor*. Furthermore, o_0 is the *head* and o_n is the *tail* of the CR. Please note that a CR is itself immutable, while the corresponding object path may change in the course of time due to a changing instance variable on the path.

Just like any reference in a statically typed language, (a) a CR has a type, (b) it can be compared to other CRs, and (c) methods can be invoked on it. We define the *static type* of a CR to be the static type of its tail. It might seem to be straightforward to equally define the dynamic type of a CR. However, we decided to call the dynamic type of the tail the *temporary type* of a CR, because this type may change as a side effect of a field update. Downcasts to the temporary type of a CR are *disallowed* in our model because references that are typed to the temporary type may become invalid after a field update. This issue is further discussed in section 5.

Let us now consider the identity semantics in the context of compound references. The question is: Under which conditions are two compound references $s \equiv o<-v_1<-...<-v_n$ and $t \equiv p<-w_1<-...<-w_m$ with their corresponding object paths $o_0<-o_1<-...<-o_n$ and $p_0<-p_1<-...<-p_m$ considered identical?

There are at least three possible answers:

- **Head identity:** $s == t :\Leftrightarrow o == p$.
- **Tail identity:** $s == t :\Leftrightarrow o_n == p_m$.
- **Path identity:** $s == t :\Leftrightarrow n = m$ and $o_i == p_i$ for $i = 0, \dots, n$.

Head identity seems to be awkward because references would be considered identical that are - in general - not even of the same type (the static type of a CR is the static type of its tail). For example, CRs to the account resp. to the address of the same person would be considered identical. Path identity, on the other hand, seems to be too restrictive. Recall the account example. If Jack and Sally share an

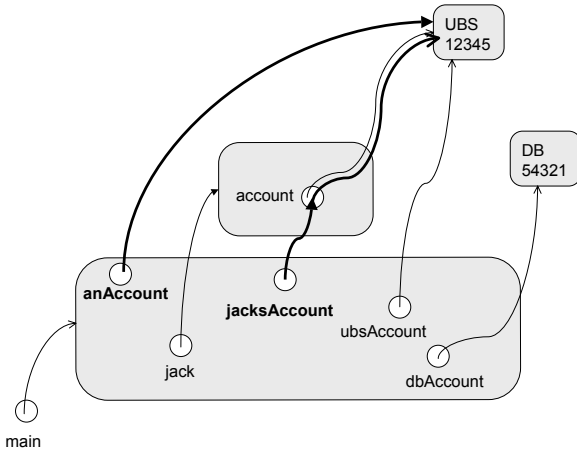


Figure 11: State before changing Jack's account

account, then we want Jack's account, i.e., the CR `jack<-account`, to be identical to Sally's account, i.e., to the CR `sally<-account`. Hence, tail identity seems to be the only reasonable identity semantics. For this reason, we define two CRs to be identical if and only if their tails are identical as defined above.

Finally, let us consider the method call semantics on a CR. If a method implemented by an object `o` is called via a CR on `o`, the value of the implicit `this` parameter is actually the CR and not `o`. That is, if during the execution of the method the object `o` passes itself to another object, it actually passes the CR by which the method was called. For convenience, we add some syntactic sugar: A method call `(this<-a).m()` is abbreviated to `a<-m()`.

For illustration, reconsider the `TextJustifier` implementation in Fig. 9. Truly incremental modification would mean to implement only `test()` and `action()` since these are the only methods, the semantics of which should be specific when used in the context of a `TextJustifier`. The question is now, how would then the specific iteration step semantics implemented by `TextJustifier::it.action()` get integrated into the iteration process which is performed by the `doAll()` method called on the instance variable `it` of a `TextJustifier`? Here is where the interplay between field methods and CRs becomes relevant. If methods are dispatched via a compound reference, field methods override corresponding methods of successive objects. In more detail, the semantics is as follows. Let `myref` \equiv `o<-v1<-...<-vn` be a CR with object path `o0<-o1<-...<-on` and `m()` be a method of the static type of `vn`. Furthermore, let `i` be the lowest index such that the class of `oi` contains a field method `vi+1...vn.m()` (a normal method is regarded as a field method with empty prefix). Then a method call `myref.m()` will be dispatched to the field method `vi+1...vn.m()`.

The implementation in Fig. 13 illustrates the interplay of CRs and explicit field methods. Within

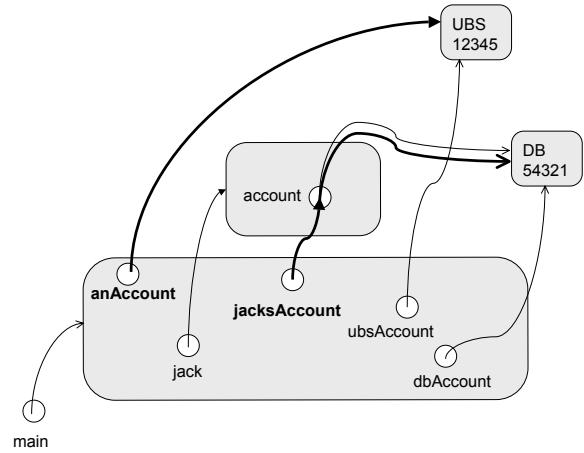


Figure 12: State after changing Jack's account

```
class TextJustifier {
  private TreeIterator it;
  private void it.action(Object x) { ... }
  private boolean it.test(Object x) { ... }
  public void justify() {... it<-doAll(...); }
}
```

Figure 13: Explicit redirected field methods in `TextJustifier`

`TextJustifier::justify()`, the method `doAll()` is not called directly on `it`, but rather via the compound reference `this<-it`. Consequently, subsequent calls to `action()` and `test()` that are made within the control flow of `TreeIterator::doAll()` will be dispatched to `TextJustifier::it.action()`, respectively `TextJustifier::it.test()`.

With the implementation in Fig. 13, the fact that `TextJustifier` uses and even customizes an `TreeIterator` is completely hidden from clients and subclasses of `TextJustifier`. It is not required for overriding methods to respect the visibility of the overridden methods, because `TextJustifier` is not a subtype of `TreeIterator`. Without any further code modification, it would also be possible to choose an iteration strategy at runtime (cf. subsection 2.3) by simply assigning a new iterator object to `it`. Thus, the implementation in Fig. 13 actually realizes a composition of `TextJustifier` and `TreeIterator` functionalities that supports overriding, transparent redirection, dynamic polymorphism, without subtyping and acquisition.

So far, compound references to an aggregated object referred to by a field `f` are only explicitly created by the aggregating class containing `f` before a method call (cf. `f<-m()`). That is, the scope of the composition features mentioned above (overriding, transparent redirection, and dynamic polymorphism) is an individual method call via an explicitly created CR to `f`. This is different with fields that are declared with

```
// rf is a redirected field
private void rf.m() { field<-m(); }

// f is a non-redirected field
private void f.m() { field.m(); }
```

Figure 14: Field methods and field redirection

```
class TextJustifier {
    private redirect TreeIterator it;
    private void it.action() { ... }
    private boolean it.test(Item x) { ... }
    public void justify() { ... it.doAll(...); }
}
```

Figure 15: TextJustifier with field redirection

the modifier `redirect`. Implicit field methods of a field that is annotated with the `redirect` keyword have a different semantics: Instead of simply forwarding the call to the field object, they first implicitly create a CR to that field and call the method on the created CR. Fig. 14 shows the difference between the default implementation of implicitly available methods of a redirected field, `rf` and that of a non-redirected field `f`. For illustration, a version of `TextJustifier` with `it` declared as a `redirect` field is given in Fig. 15.

3.3 Field Acquisition

Field acquisition is another step on the road from object composition to inheritance. Orthogonal to the other modifiers, the `acquire` modifier can also annotate a field declaration. The intuitive semantics is that the features available in the field become an inherent part of the aggregating class. A class `C` with an acquired field `f` of type `F` implicitly contains a method `m()` for every public method `m()` of `F`. The method `m()` retains its signature as declared in `F` and its visibility in `C` is `public`. The semantics of the implicit field methods remains the same as with non-acquired fields, except that they are now provided in the interface of `C`. For illustration, consider the example in Fig. 16. Although `EmptyFilterStream` does not itself implement `write(int)` or `write(int[])`, these methods can be invoked on `efs` – an instance of `EmptyFilterStream` – due to the declaration of the instance variable `stream` as an acquired field.

Acquired implicit methods can also be replaced by explicitly programmed methods with the same signature. For the sake of uniformity and in order to facilitate changing of a given composition semantics by means of changing the modifiers of an instance variable, the prefix notation has to be used when explicitly overriding acquired methods. For illustration, consider the sample code in Fig. 17. The class `BufferedOutputStream` acquires both `write` methods from its acquired field `stream` and overrides them to add buffering.

Note that the `bos.write(array)` call in the client code in Fig. 17 only displays “... buffering 5 ints ...” on the screen. The fact that no message “... buffering a single int ...” appears on the screen suggests

```
class OutputStream {
    public void write(int b) {
        System.out.println("Hello from write(int )");
    }
    public void write(int[] b) {
        System.out.println("Hello from write(int[] )");
    }
}

class EmptyFilterStream {
    acquire private OutputStream stream = new OutputStream();
}

class Client {
    static public void main(String[] args) {
        EmptyFilterStream efs = new EmptyFilterStream();
        int[] array = ...;

        efs.write(3);
        // "Hello from write(int )" appears

        efs.write(array);
        // "Hello from write(int[] )" appears
    }
}
```

Figure 16: EmptyFilterStream with acquired fields

that the overridden `write(int)` method as implemented in `BufferedOutputStream` is not invoked, although, at this point the `write(int)` method of the underlying `fout` stream will actually be called 5 times (since the buffer is already full, the overridden field method will be called for both the buffer and the `int` array passed as a parameter). However, “... buffering a single int ...” is not displayed because neither `stream` is a redirected field, nor are the calls `field.write(buffer)` and `field.write(b)` made via a compound reference `this<-stream`. Therefore, the calls to `write(int)` from within `field.write(...)` escape the override by the `BufferedOutputStream`. This corresponds to the *broken delegation problem* discussed in Sec. 2. In this case, this is indeed the desired semantics, i.e., redirection is actually not desired. Once the buffer is full, we want to flush buffer’s content and the integers to be written immediately to the underlying data sink. Hence, we indeed want to escape buffering.

However, there might be cases, when we want all calls occurring within the control flow of a call to an overridden acquired method of an object `outer` to be also dispatched to `outer`. If field acquisition is combined with field redirection, we obtain a perfect solution for this composition requirement. In the version of `BufferedStream` presented in Fig. 18, where `stream` is declared to be acquired and redirected, it suffices to do the buffering in the `write(int)` method, because calls to `write(int)` in the `write(int[])` method are automatically redirected to the buffering method.

One important restriction is imposed on field acquisition: We allow every class to have at **most one field acquisition**. Otherwise we would have to take charge of all those annoying multiple inheritance conflicts. However, due to

```

class OutputStream {
    public void write(int b) { ... }
    public void write(int[] b) {
        for (i = 0; i < b.size(); i++) write(b[i]);
    }
}

class BufferedOutputStream {
    acquire private OutputStream stream;
    int[] buffer; int current;

    public BufferedOutputStream(out) {
        stream = out;
        buffer = ... ;
    }

    public void stream.write(int b) {
        System.out.println("... buffering
            a single int ... ");
        if (buffer.notFull()) buffer[current++] = b;
        else {
            field.write(buffer);
            field.write(b);
            current = 0;
        }
    }
    public void stream.write(int[] b) {
        System.out.println("... buffering "
            + b.size + "ints ...");
        if (buffer.size() >= current + b.size()) {
            System.arraycopy(b,0,buffer,current,b.size);
            current += b.size;
        } else {
            field.write(buffer)
            field.write(b);
            current = 0;
        }
    }
}

class Client {
    static public void main(String[] args) {
        FileOutputStream fout =
            new FileOutputStream(aFileName);
        BufferedOutputStream bos =
            new BufferedOutputStream(fout);
        int[5] array = ...;

        bos.write(3);
        // "... buffering a single int..."
        // appears on the screen
        ...

        //assume that buffer is full at this point

        bos.write(array);
        // "... buffering 5 ints ..."
        // appears on the screen
    }
}

```

Figure 17: Overriding acquired fields

```

class BufferedStream {
    acquire redirect private OutputStream stream;
    public void stream.write(int b) {
        ...do buffering...
        field.write(b);
    }
}

```

Figure 18: Overriding and redirecting acquired fields

the fact that we can (a) do overriding and redirection for multiple fields, and (b) combine multiple classes by means of organizing them in an acquisition chain, this is no grave limitation.

There is a second restriction that we need to make. Due to subtype polymorphism, an instance of a subtype of `OutputStream` may be assigned to the `stream` instance variable. This subtype may contain methods that are not available in `OutputStream`. These methods should not be overridden, because this might lead to unexpected or unsound results: The result might be unexpected because the author of the overriding method does not know about the existence and semantics of the overridden methods. The result might also be unsound, because the overriding method may have a signature that is not compatible with the signature of the overridden method (e.g., has a different return type, see also [18]). For this reason, we make the following restriction: A field method overrides a method defined in a field type if and only if it is already defined in the *static* type of the field type.

The addition of the `acquire` feature into the model, has further enriched the range of composition semantics between the classes `C` and `F` that the programmer can express. Used in isolation, `acquire` enables `C` to transparently forward “services” to `F`, whenever it needs to do so, in order to satisfy a request from an external client. On the other hand, combining `redirect` and `acquire` yields a mechanism for incremental modification that mimics the code reuse provided by inheritance or delegation.

3.4 Subtyping

One thing is still missing on the road to inheritance/delegation-based composition: Field acquisition does not imply subtyping. A class can explicitly declare to be a subtype of a number of other types via a `subtypeof` clause. Declaring a class `C` as a subtype of a type `T`, requires that `C` has to either implement all methods that are defined in `T` or be abstract. In contrast to Java’s `implements` clause, in our model both interfaces *and* classes may appear on the right hand side of a `subtypeof` clause⁸.

Declaring a class `D` to be a subtype of another class `C` means that `D` implements the interface of `C`, but it does not mean that the implementation of `C` can automatically be used for the realization of the corresponding methods in `D` - `D` does

⁸It is still possible to use traditional inheritance with `extends`.

```

class OutputStream { ... }
class FileOutputStream extends OutputStream { ... }

class FilterStream subtypeof OutputStream {
  acquire redirect protected OutputStream stream;
}
class BufferedOutputStream extends FilterStream {
  public void stream.write(in b) {
    ... do buffering ... ;
    field.write(b);
  }
}
class CompressedOutputStream extends FilterStream {
  ...
}

```

Figure 19: OutputStream in our model

not automatically acquire the state and method implementations of *C*. However, *D* can still make use of the behavior defined in *C*, if this is desired, by declaring a field of type *C* with modifiers `acquire` and `redirect`. This is an important step towards a better separation of types and classes. Decoupling subtype declaration from implementation reuse solves e.g., the last drawback of the decorator approach explained in section 2.2.

For illustration, the “complete” implementation of the stream example from Fig. 2 in our model is given in Fig. 19. Compare this to the simulation of redirect semantics in Fig. 3.

3.5 Field Navigation

If the object referred to by a field represents a facet that should be visible to clients, we would like to have this fact made explicit in the declaration of the field, rather than relying on the presence of appropriate getter methods. For serving this purpose, a field can be made navigable by annotating it with the `navigable` modifier. For example, we could annotate the `account` field of `Person` as navigable, as shown below. This allows clients to directly navigate to this part of the object by retrieving a compound reference to that part. This is illustrated below by having the client of the `Person` object `p` retrieve a CR to `p`’s `account` and store it in `a`. Technically, the declaration of a field as navigable can be seen as short-cut for the corresponding getter methods discussed above.

```

class Person {
  private navigable Account account;
}
Person p = ...
Account a = p<-account;

```

Note that declaring a field as navigable does not imply that clients can directly change the field. The possibility to navigate to a field becomes part of the class interface, similar to a getter method that returns the current value of a field. Actually, the navigable composition semantic flavor discourages rather than supports breaking encapsulation. Exporting a CR to an instance variable to external clients as part of the interface of a class *C* can also be simulated by declaring a redirect field to be public. However, this breaks the

encapsulation of *C*: clients can freely change the value of the reference. This is not possible with navigable references.

The inverse navigation operation is provided by a CR *reduction operator* that can be used to access previous objects on the object path of a CR. A reduction `<AType> myref` on a compound reference `myref ≡ o<-v1<-...<-vn` creates a new compound reference `o<-v1<-...<-vn-1`. The reduction succeeds if the type of the shortened compound reference (that is, the static type of the `vn-1` variable) is a subtype of *AType*. If the length of the source path is two, a primitive reference to `o` is created. Since it is not statically known whether an account reference is a compound reference via a person, a CR reduction has to be checked at runtime. The reduction operation is in a way similar to type downcasts in languages like Java. In the `Account` example, we could reduce a compound reference to an `Account` to `Person`:

```

Person p = ...
Account a = p<-account;
Person p2 = <Person> a; // ok, checked at runtime
(p == p2) // true

```

4. EVALUATION OF THE MODEL

After having introduced the individual steps on the road from object composition to inheritance, it is now time to show how the problems discussed in Sec. 2 are addressed in our model. The key to addressing these problems is the availability of rich linguistic means to express a variety of composition flavors by simply decorating object references with composition properties. To support the discussion, Fig. 20 introduces graphical notations for some of the most relevant composition flavors between two classes *C* and *F* that can be expressed with the model⁹. Please note that these notations do not address overriding because in our model overriding is implicitly available by means of field methods and need not be explicitly turned on or off.

Let us start with the the composition flavor (b) - in the middle of the road between object composition and inheritance. A composition *C(F)* with this flavor shares with inheritance overriding with late binding. This is not available with object composition. On the other side, such a flavor shares with object composition dynamic polymorphism as well as lack of both acquisition and subtyping. The latter two features are inseparable from inheritance/delegation, though. The discussion of the `TextJustifier` example in Sec. 2 indicated that such a mixture of features might indeed be needed and that the lack of linguistic means to express it forces the programmer in an object-oriented language to simulate the same semantics by means of complex, unclear architectures that are fragile with respect to requirement changes.

In the previous section, we have modeled the same composition scenario in our model. The implementation of the desired composition semantics between `TextJustifier` and the `TreeIterator` hierarchy is as simple as the code

⁹This list is not intended to cover all possible combinations of composition features, but only those that are relevant for evaluating the model with respect to the issues discussed in Sec. 2.

- (a) Object composition
- (b) Object composition with redirection
- (c) Object composition with acquisition
- (d) Object composition with acquisition and redirection
- (e) Object composition with acquisition, redirection and subtyping

Figure 20: Graphical notations for different composition flavors

in Fig. 15 and the design as clear as the class diagram presented in Fig. 21, which is as simple as the inheritance-based design in Fig. 5. However, in contrast to the inheritance-based design, with the CR-based solution (1) several iteration strategies can be chosen dynamically, (2) the iteration functionality does not pollute the interface and implementation of `TextJustifier`, and (3) the conceptual view that a `TextJustifier` is not a special kind of `TreeIterator` is preserved. The latter are features that were indeed also supported by the architecture based on object-composition presented in Fig. 6 and 7. However, our design does not share the complexity of the designs presented in Fig. 6 and Fig. 7.

The complexity of designs that simulate non-standard composition flavors was only one of the problems that we identified in Sec. 2. The second and more important problem, was that different composition flavors were modeled by different architectures. In the following, we demonstrate, that this problem is avoided in our model, by reconsidering the text justifier and stream example from Sec. 2.

The design in Fig. 21 encodes a composition with redirection, overriding and dynamic polymorphism. Assume, we also want to have acquisition. In our model, we would simply add the `acquire` modifier to the declaration of `it`. The class diagram in Fig. 21 remains the same, except for replacing the current `it` link with link (d) in Fig. 20. On the contrary, with the designs in Fig. 6 and Fig. 7 we would have to change `TextJustifier` to implement all methods in the interface of `TreeIterator` by forwarding these methods to `it`. If we additionally want to have `TextJustifier` be a subtype of `TreeIterator`, we would again merely have to replace the `it` link with the link (e) in Fig. 20. The resulting design would still encode a different composition flavor as compared to the inheritance based composition in Fig. 5, since (1) we still have a composition that supports dynamic polymorphism and (2) `TextJustifier` would not inherit the state of `TreeIterator`.

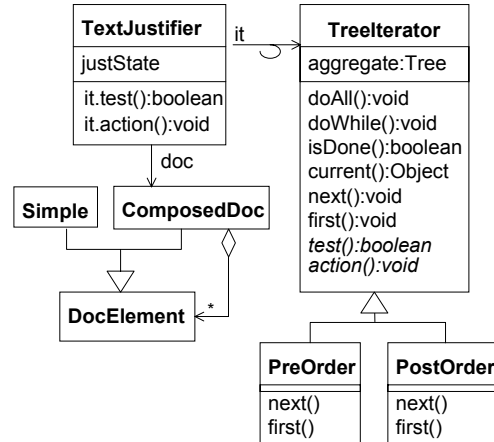


Figure 21: TextJustifier using compound references

A similar seamless transition from one composition flavor to the other was observed when we modeled different flavors of `BufferedOutputStream` in Fig. 17, Fig. 18, and Fig. 19. Here we started with a flavor that is closer to the inheritance end of the composition flavor spectrum: Object composition with acquisition semantics. We then added redirection and subtyping in two separate steps.

Another important feature of our model which makes it superior to standard composition models is the fact that a class can simultaneously reuse and adapt the functionality of several other classes without suffering from the known multiple-inheritance conflicts:

- Naming conflicts: Different methods with the same name are inherited.
- Repeated inheritance (a.k.a. diamond inheritance): The same class is inherited twice indirectly, for example D is a subclass of B and C, both of which are subclasses of A. Is a single copy of A shared by B and C or are there two copies? What happens to methods that are overridden in B and C in the first case? Which copy of A do clients of D see in the latter case?

Different mechanisms have been developed to cope with these problems, but avoiding a problem is certainly better than fixing it. Due to our naming definition for field methods (prefixing it with the name of the attribute), we have no naming conflicts. Problems related to repeated inheritance do not occur either, because every compound reference induces a unique path for message dispatch. The question whether we have a shared or replicated “parent” boils down to assigning the same, respectively different instances of the aggregated class to the corresponding attributes.

Finally, we would like to bring up navigable fields in this evaluation, because they foster another spectrum of relationships not discussed so far. Industrial component models like

COM [6] and CCM [23] have the notion of independent interfaces or facets that a component exposes and that can be retrieved by special navigation methods. Design patterns such as *extension interface* [26] or *extension object* [14] propose architectures to allow a class to export multiple unrelated interfaces à la COM and CCM without employing inheritance or subtyping. However, as also acknowledged by the authors, the proposed patterns incur increased design and implementation effort, e.g., navigation infrastructure that is of no functional use but necessary to retrieve the facets [26], and increased client complexity [14, 26]. This critique is in the vein of our discussion in the motivation section.

Navigable fields present an elegant approach to modeling classes that export several unrelated interfaces. A class C exports the interfaces of all navigable fields. This is explicitly declared in the class' interface. This export involves no interface bloat because C 's interface does not itself contain the methods of the exported interfaces. This is in contrast to a class in Java implementing several different interfaces. In contrast to the extension interface and extension object patterns, the feature of exporting several unrelated interfaces is built into the language and integrated with static type checking. The relationship that the exported interfaces are “facets” of the behavior of the exporting class is explicit in the exporting class' interface. The same relationship is not explicit in the design of the extension interface and extension object patterns as also indicated by Gamma [14].

5. ADVANCED ISSUES

Until now, compound references have been introduced in a rather informal way. In this section, we will provide more details. In particular, we will show that our model is type safe. Type safety is threatened by subtle combinations of compound references and subtype polymorphism.

In section 3.2, the static and the temporary type of a CR have been defined. We have argued that type casts to the temporary type of a CR should not be allowed because the temporary type may change in the course of time. Enforcement of this invariant is trivial for explicit type casts in the program code. However, there are situations when we have to cast a CR to its temporary type: Consider a class A with a field b of type B . At runtime, an instance of $BSub$, a subtype of B , is assigned to b and A makes a call $b \leftarrow m()$ to this object. The method $m()$ of B is overridden in $BSub$. This means that we execute the method $m()$ of $BSub$ and the actual value of $this$ is the CR $a \leftarrow b$ with static type B . However, the type of $this$ has to be (at least) the type of the corresponding class because otherwise features that are introduced in $BSub$ could not be called.

These casts to the temporary type are the cause that under certain conditions the naive algorithm for creating an object path $o_0 \leftarrow o_1 \leftarrow \dots \leftarrow o_n$ for a CR $o \leftarrow v_1 \leftarrow \dots \leftarrow v_n$, namely $o_0 = o$ and $o_i = o_{i-1}.v_i$, fails. Fig. 22 shows three different scenarios that lead to type errors if the naive algorithm is employed. In the first scenario, the algorithm fails because the new object in b no longer contains a field c when the $q()$ method of o is called. In the second one, the `Other` instance expects its reference to be of type `BSub`, but the

new value for b is an instance of B . In the last scenario, the CR that constitutes $this$ during the execution of `BSub.m()` is changed while the method is still on the call stack.

Some of these problems also occur in delegation-based systems and different solutions have been proposed (see [19] for an overview). However, most of these approaches do not really fit in our model, because *all* fields are the potential targets of CRs, so that trivially type safe restrictions like requiring the new value of a field to be a subtype of the *dynamic* type of the previous object are not practicable.

Instead, we present a dispatch algorithm that guarantees static type safety while preserving the unrestricted programming model. The main idea of our approach is as follows: On every cast to the temporary type of a CR, we store the current field value in the CR. This original value is used whenever the current value would lead to a type error.

For the definition of this algorithm, we choose a recursive representation of CR: A CR is either a primitive reference or a pair $parent \leftarrow v$ such that $parent$ is a CR and v a field name. We call a CR that has been casted to its temporary type (or a supertype of the temporary type that is not a supertype of the static type) a *critical CR*. A critical CR is a triple $parent \leftarrow v \mid s$ such that s is the stored field value. Please note that due to the recursive construction $parent$ may already be a critical CR. A non-critical CR is converted to a critical CR by storing the current value of the field in s . This “conversion” (think of the CR as being passed “by value”) takes place whenever the CR is subject to an implicit downcast to its temporary type. The decision whether the current or the stored field value is used is based on an additional parameter, the *requested type reqType*, that defines which type is expected in the actual context. For a CR `ref`, $reqType$ is the declared type of `ref`. In the following, C_v denotes the class in which the field v is defined. For non-critical CR, the object path is created as follows:

```

objectPath(parent ← v, reqType) :=
    objectPath(parent, C_v) ← tail(parent ← v, reqType)

tail(parent ← v, reqType) :=
    tail(parent, C_v).v

```

Except for the additional *reqType* parameter, this algorithm is equivalent to the non-recursive description $o_i = o_{i-1}.v_i$. In the critical case, the *reqType* parameter comes into play:

```

tail(parent ← v | s, reqType) :=
    if tail(parent, C_v).v instanceof reqType
    then tail(parent, C_v).v
    else s

```

An induction proof on the length of CR shows that this algorithm preserves type safety. It suffices to proof that the type of the object that is returned by the *tail* function is always a subtype of *reqType*. For CRs of length one (that is, primitive references) the claim holds because the base

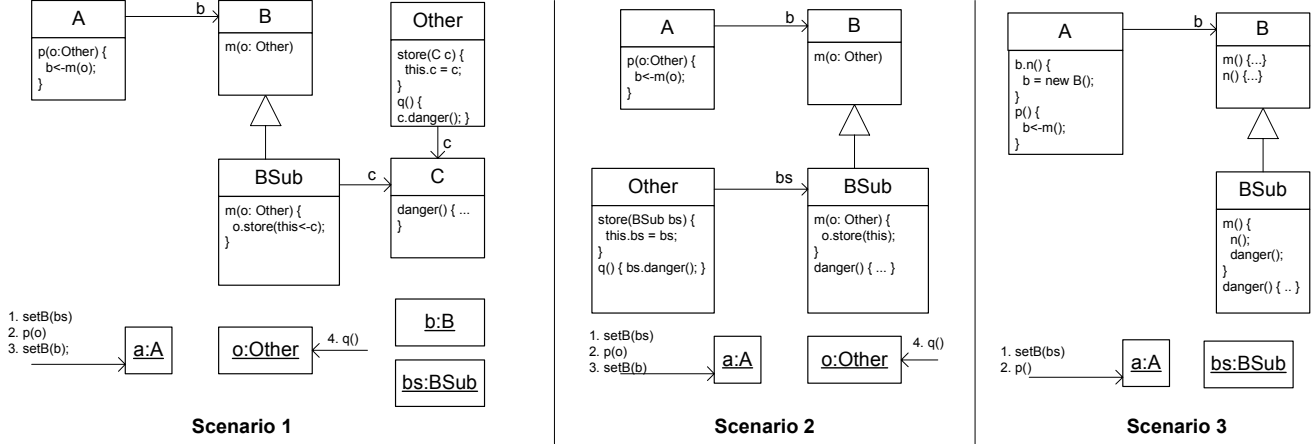


Figure 22: Scenarios that threaten type safety

language (without CR) is assumed to be statically type safe. Let `ref` be a CR with length n and declared type T . Then `ref` may be a critical or a non-critical CR.

1. **ref is non-critical.** Let `ref` \equiv `parent<-v`. T = `reqType` is a supertype of the static type of `ref` because otherwise `ref` would be critical. By induction hypothesis, the type of `tail(parent, Cv)` is a subtype of C_v , so that the field value v , o_v can be safely retrieved. Subtyping guarantees that this object is a subtype of the declared type of v , and therefore also a subtype of `reqType`.
2. **ref is critical.** Let `ref` \equiv `parent<-v | s`. By induction hypothesis, the type of `tail(parent, Cv)` is a subtype of C_v , so that the field value v , o_v can be safely retrieved. The `if` statement guarantees that `ov` is returned if and only if it is an instance of `reqType`. If this is not the case, `s` is returned, so we have to show that `s` is a subtype of `reqType`. This is assured by the rule that a critical CR is created and initialized whenever a non-critical CR is casted to its temporary type because this implies that during the assignment of `ref` `s` has been (and is) a subtype of the static type of `ref`.

Applied to the scenarios in Fig. 22 this means that in scenario 1, the reference `c` in `o` is critical, and the call to `danger()` in `Other::q()` is dispatched via the original object path `bs<-c`. This is also the case in scenario 2 and 3: The reference `bs` in `o` resp. `this` in `bs` is critical and the `danger()` call is dispatched via the original object path `a<-bs`.

We have shown that our dispatch algorithm renders the model statically type-safe. However, a price has to be paid: Although this happens only under very special conditions, it is an undesirable complication of the model that the invariant that all calls to a CR are always dispatched to the current object of the corresponding field does not hold anymore. We think that the reason for these problems is the

existence of a “magic triangle” between (a) type safety, (b) expressive power, and (c) easy semantics. In our proposal, we put the emphasis on (a) and (b) and got some problems with (c).

6. RELATED WORK

Delegation appeared first in untyped, prototype-based languages [20]. The most prominent example in this category is SELF [32]. As shown in Table 1, delegation includes all composition properties simultaneously; applying individual properties independently is not explicitly supported.

More recent proposals have been proposed to restrain the extreme flexibility offered by SELF and a number of related proposals by embedding delegation in a statically typed language. The DARWIN model [18, 19] combines delegation and static inheritance in a statically typed language. DARWIN already incorporates a limited variant of composition property separation: Besides delegation and inheritance, DARWIN also has the notion of *consultation*, which, in our terminology, corresponds to delegation without redirection.

GENERIC WRAPPERS [4] support a restricted variant of delegation: Once a “wrappee” is assigned to a “wrapper”, the wrappee is fixed. In our terminology, this corresponds to delegation with “semi-dynamic” polymorphism (parent fixed at runtime), and in our model would be expressed by declaring the corresponding attribute as `final`. Büchi and Weck [4] emphasize the importance of being able to dynamically cast a wrapper to the dynamic type of its wrappee (*transparency*). In our model, this could be achieved by allowing explicit dynamic casts to the temporary type of a CR, which is not problematic, when the attributes are annotated `final`. However, further details on this aspect have been left out of the scope of this paper.

GBETA [11] also has a number of dynamic features that are related to delegation. Like in GENERIC WRAPPERS, parents in GBETA are fixed at runtime. GBETA also allows dynamic behavior additions to objects that preserve object identity,

for example a statement like `aClass##->anObject##` adds the structure of `aClass` to `anObject`. Another delegation-based approach is described in [28]. Steyaert and De Meuter propose a variant of delegation in which a class has to anticipate all its possible extensions in order to avoid certain encapsulation problems.

Compared to these approaches to supporting delegation in a statically typed language, delegation, in our model, comes out as a special mixture of composition properties, among many other possible mixtures. In addition, our model is more flexible in that in contrast to the aforementioned approaches, objects do not have a *single special* parent attribute. In our model it is possible to override and redirect *multiple arbitrary* attributes.

PREDICATE OBJECTS [10], RONDO [21], and the CONTEXT RELATIONSHIP [27] allow the programmer to express certain kinds of context-dependent facets of an object by explicit linguistic means. The composition of the basic behavior of an object and its facets obeys delegation semantics in [10, 27], and some form of mixin-based inheritance in [21]. Our model shares with these approaches the support for a two-dimensional incremental modification: (1) vertically by means of inheritance in our model, RONDO, and CONTEXT RELATIONSHIP, respectively by means of delegation in PREDICATE OBJECTS, and (2) horizontally by means of an advanced form of delegation in [10], an advanced form of inheritance that supports the static/dynamic polymorphism property in [21, 27], and by means of CRs in our model. However, in [10, 21, 27] the composition flavors in both axes are built in; individual composition properties are not explicitly available for on-demand combination.

MIXIN-BASED INHERITANCE [7, 12, 2] is an enrichment of normal inheritance with the static polymorphism feature of Tab. 1. In contrast to the normal inheritance, the `super` pseudo-variable of a subclass is not bound to a certain base class when the subclass is defined. Rather, there is an explicit composition stage, where `super` is statically bound to a composition-specific superclass. In this way, the same subclass (*mixin*) can be statically applied to several base classes. However, inheritance enhanced with static polymorphism is the only composition flavor supported.

JIGSAW [8] improves the modularity of the original mixin-based inheritance [7] by providing a suite of language operators that independently control several roles that classes play in standard languages such as combination of features, modification, encapsulation, name resolution, and sharing. This untangling of class composition semantics is in its core very similar to our untangling of standard composition semantics. The motivation for undertaking these untanglings is different, though. The main focus in JIGSAW is on fine-grain control over the visibility of the features from the individual modules in a composition, to allow mixins, multiple inheritance, encapsulation, and strong typing to be combined in cohesive manner.

The flexible control over the method dispatch via filters attached to an object complemented by the ability to define

different facets of an objects in so-called **internal** and **external** objects supported by the COMPOSITION FILTERS approach [1] can probably also be used to simulate some of the flavors of composition semantics that can be expressed in our model. Still, there are important differences between the two models. First, the COMPOSITION FILTERS approach lacks a static type system. Second, different flavors of composition semantics need to be manually implemented in different dispatch filters. This might turn out to be a tedious and error-prone activity, especially if several internals and mixtures of composition properties are involved. In contrast, the specification of the desired semantics is more declarative in our model. Third, it is not obvious how redirection semantics could be "programmed" with dispatch filters.

Our notions of field methods and field navigation share some commonality with the **as-expressions** of the POINT OF VIEW NOTION OF MULTIPLE INHERITANCE [9] in that they allow to adapt and combine multiple classes without suffering from multiple inheritance conflicts. However, due to the use of object rather than class composition, our approach is more flexible when coping with issues such as sharing and duplicating the features of common parents, typical for approaches to multiple inheritance.

7. SUMMARY AND FUTURE WORK

In this paper, we showed that the traditional object-oriented composition mechanisms, object composition and inheritance/delegation, are frequently inappropriate to model non-standard composition scenarios. Non-standard composition semantics are simulated by complicated architectures that are sensitive to requirement changes and cannot easily be adapted without invalidating existing clients. This unsatisfactory situation is due to the fact that the combination of composition properties supported by each mechanism is fixed in the language implementation and individual properties do not exist as abstractions at the language level.

We proposed compound references as a new and powerful abstraction for object references. On this basis, we were able to provide explicit linguistic means for making individual composition properties available and to allow the programmer to express a seamless spectrum of composition semantics in the interval between object composition and inheritance. The model is statically type-safe and makes object-oriented programs more understandable, due to explicitly expressed design decisions, and less sensitive to requirement changes.

Two issues that have already been worked out but have been omitted due to space reasons are as follows: First, there is a well-known conflict between delegation and method header specialization [4, 19]. Second, the concept of abstract classes and abstract methods is also useful for object-based overriding. Statically safe solutions to both problem are proposed in [24].

There are some areas of future work. First, the fine-grained scale between object composition and inheritance renders the common visibility modifiers `public` and `protected` too coarse, so that a more sophisticated visibility concept is desirable. Such a refined visibility concept may also solve en-

capsulation problems as described in [28]. Another interesting area is to investigate the space of possible composition property combinations for invalid combinations, which need to be rejected at compile-time. Finally, an interesting extension of the CR concept would be to also allow CRs to dictionary entries, so that the dictionary keys take the roles of field names, and the corresponding values the role of field values.

8. ACKNOWLEDGMENTS

We thank Frank Buschmann, Lutz Dominick, Bernd Freisleben, Stephan Herrmann, Günter Kniesel, Markku Sakkinen and the anonymous reviewers for helpful comments.

9. REFERENCES

- [1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Programming*. Springer, 1993.
- [2] D. Ancona, G. Lagorio, and E. Zucca. Jam - a smooth extension of Java with mixins. In *Proceedings ECOOP 2000*, pages 154–178. LNCS, Springer, 2000.
- [3] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
- [4] M. Büchi and W. Weck. Generic wrappers. In *Proceedings of ECOOP 2000, LNCS 1850*, pages 201–225. Springer, 2000.
- [5] K. Beck, M. Fowler, and J. Kohnke. *Planning Extreme Programming*. Addison-Wesley, 2000.
- [6] D. Box. *Essential COM*. Addison-Wesley, 1997.
- [7] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices 25(10)*, pages 303–311, 1990.
- [8] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings of IEEE Computer Society International Conference on Computer Languages*, pages 282–290, Washington, DC, April 1992. IEEE Computer Society.
- [9] B. Carré and J. Geib. The point of view notion for multiple inheritance. In *Proceedings OOPSLA/ECOOP '90*, pages 312–321. ACM SIGPLAN Notices, vol. 25 no. 10, 1990.
- [10] C. Chambers. Predicate classes. In W. Olthoff, editor, *Proceedings ECCOP '93*, LNCS 707, pages 268–297. Springer, 1993.
- [11] E. Ernst. *gbeta - a language with virtual attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.
- [12] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *ACM Symp. on Principles of Programming Languages '98*, pages 171–183, 1998.
- [13] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [14] E. Gamma. Extension object. In R. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design*, pages 79–88. Addison-Wesley, 1998.
- [15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.
- [16] W. Harrison, H. Ossher, and P. Tarr. Using delegation for software and subject composition. Technical Report RC 20946(92722), IBM Research Division T.J. Watson Research Center, Aug 1997.
- [17] F. J. Hauck. Inheritance modeled with explicit bindings: An approach to typed inheritance. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, 1993.
- [18] G. Kniesel. Type-safe delegation for run-time component adaptation. In R. Guerraoui, editor, *Proceedings of ECOOP '99*, LNCS 1628. Springer, 1999.
- [19] G. Kniesel. *Dynamic Object-Based Inheritance with Subtyping*. PhD thesis, University of Bonn, Institute for Computer Science III, 2000.
- [20] H. Liebermann. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, 1986.
- [21] M. Mezini. Dynamic object evolution without name collisions. In *Proceedings ECOOP '97, LNCS 1241*, pages 190–219. Springer, 1997.
- [22] M. Mezini. *Variational Object-Oriented Programming Beyond Classes and Inheritance*. Kluwer Academic Publisher, 1998.
- [23] Object Management Group. *CORBA Components Final Submission*. OMG TC Document orbos/99-02-05, 1999.
- [24] K. Ostermann. Object-Oriented Composition: An Analysis and a Proposal. Master's thesis, Universität Bonn, Institut für Informatik III, 2000.
- [25] M. Sakkinen. Disciplined inheritance. In *Proceedings ECOOP '89*, pages 39–56. Cambridge University Press, 1989.
- [26] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture Vol. 2*. Wiley, 2000.
- [27] L. Seiter, J. Palsberg, and K. Lieberherr. Evolution of object behavior using context relations. In D. Garlan, editor, *Proceedings of the 4th ACM SIFSOFT Symposium on Foundations of Software Engineering*, pages 46–56. ACM Press, 1996. Software Engineering Notes 21(6).

- [28] P. Steyaert and W. D. Meuter. A marriage of class- and object-based inheritance without unwanted children. In W. Olthoff, editor, *Proceedings of ECOOP '95*, pages 127–144. LNCS 952, Springer, 1995.
- [29] Sun Microsystems. Java 2 SDK Documentation. <http://java.sun.com/j2se/1.3/docs/index.html>.
- [30] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
- [31] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):439–479, 1996.
- [32] D. Ungar and R. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA '87, ACM SIGPLAN Notices 22(12)*, pages 227–242, 1987.