

Blurring the Borders between Object Composition, Inheritance, and Delegation

Klaus Ostermann¹ and Mira Mezini²

¹ Siemens AG, Munich, Germany, Klaus.Ostermann@mchp.siemens.de

² Darmstadt Technical University, 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. This paper is an appetizer³ for *compound references* and *Latte Macchiato*⁴, a Java implementation of compound references. Compound references are 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.

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., [14, 15, 12]. Also, the relationship between inheritance and object composition is carefully studied, e.g., in [8, 7]. 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* [10] 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 [16]. More recently delegation-based techniques are integrated into statically typed, class-based languages, which thus provide class-based inheritance, delegation, and object composition [9, 5, 2]. On the other side, several *mixin*-based models [3, 11, 6, 1] 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.

³ The full details are described in [13]

⁴ A public download for *LatteMacchiato* is available at <http://www.st.informatik.tu-darmstadt.de/lm/index.html>

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-involutions within B ignore redefinitions in M .
2. **Transparent redirection**: The ability to transparently redirect B 's `this` to denote $M(B)$ within the composition.
3. **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).
4. **Subtyping**: The promise that $M(B)$ fulfills the contract specified by B , or that $M(B)$ can be used everywhere B is expected.
5. **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.

| | 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

The key idea of the approach motivated 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.

1.1 An Introductory 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. 1⁵. 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. 1. 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. 2⁶ (left) 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

⁵ In a more realistic situation, one would have to apply the visitor pattern to connect `TextJustifier` and the `DocElement` 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.

⁶ In the design we assume that `DocElement` implements `Tree`

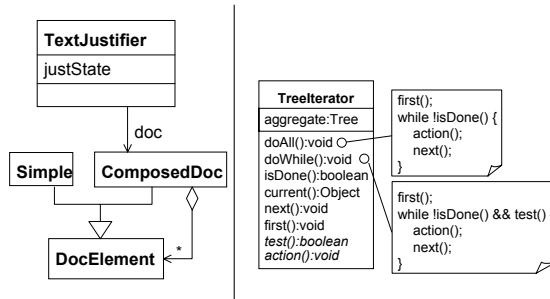


Fig. 1. Structure of text justifier and tree iterator

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. 1 with the iteration hierarchy, such that the above set of composition properties are satisfied. A feasible solution is schematically presented in Fig. 2 (right). `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 that the test and the action performed in each step of the iteration needs information from the text justifier object, which is provided via the `context` reference on the `TextJustifier`.

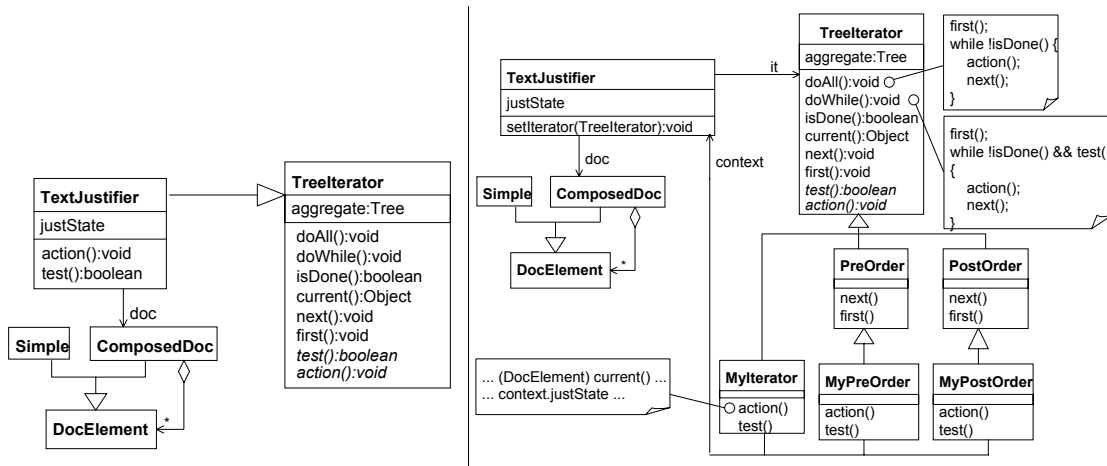


Fig. 2. Static (left) and dynamic (right) iterator composition

Obviously, the design in Fig. 2 (right) is very different from the predecessor (Fig. 2 left). 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. 2 (left), 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()`.

Let us take a quick look what the solution in our model looks like. Our language provides dedicated abstractions for each of the composition properties which has been defined in the introduction. In this case, we want a composition with redirection, overriding and polymorphism, but with neither subtyping nor acquisition semantics.

```

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

```

Fig. 3. TextJustifier with field redirection

Fig. 3 shows our solution to the text justifier example. In our model, instance variables like `TreeIterator it` can be modified with special modifiers like `redirect` or `acquire`, which augment the object composition by additional properties. The `redirect` modifier in Fig. 3 collaborates with additional abstractions called *field methods*. The methods `it.action()` and `it.test()` in Fig. 3 are examples of such field methods. The semantics is that all method calls to `action()` and `test` inside `TreeIterator` are redirected to their corresponding field methods in `TextJustifier`, if these methods are called in the scope of a method call originating in `TextJustifier`, such as the `it.doAll()` call in `justify()`.

Similarly to redirection, we have modifiers for acquisition and subtyping. As already mentioned in the abstract, the full details of the approach are specified in [13]. In the last months, Michael Eichberg, a masters student, and Klaus Ostermann have been working on a Java-based implementation called *Latte Macchiato* [4]. The work is nearly finished and the preliminary result can be downloaded at <http://www.st.informatik.tu-darmstadt.de/lm/index.html>.

References

1. 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.
2. M. Büchi and W. Weck. Generic wrappers. In *Proceedings of ECOOP 2000, LNCS 1850*, pages 201–225. Springer, 2000.
3. G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP’90, ACM SIGPLAN Notices 25(10)*, pages 303–311, 1990.
4. M. Eichberg. Latte macchiato. Master’s thesis, Darmstadt Technical University, 2002.
5. 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.
6. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *ACM Symp. on Principles of Programming Languages ’98*, pages 171–183, 1998.
7. 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.
8. F. J. Hauck. Inheritance modeled with explicit bindings: An approach to typed inheritance. In *Proceedings OOPSLA ’93, ACM SIGPLAN Notices*, 1993.
9. G. Kniesel. Type-safe delegation for run-time component adaptation. In R. Guerraoui, editor, *Proceedings of ECOOP ’99*, LNCS 1628. Springer, 1999.
10. H. Liebermann. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings OOPSLA ’86, ACM SIGPLAN Notices*, 1986.
11. M. Mezini. Dynamic object evolution without name collisions. In *Proceedings ECOOP ’97, LNCS 1241*, pages 190–219. Springer, 1997.
12. M. Mezini. *Variational Object-Oriented Programming Beyond Classes and Inheritance*. Kluwer Academic Publisher, 1998.
13. K. Ostermann and M. Mezini. Object-oriented composition untangled. In *Proceedings OOPSLA ’01*, <http://www.st.informatik.tu-darmstadt.de/~ostermann/oopsla01.pdf>, 2001.
14. M. Sakkinen. Disciplined inheritance. In *Proceedings ECOOP ’89*, pages 39–56. Cambridge University Press, 1989.
15. A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):439–479, 1996.
16. D. Ungar and R. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA ’87, ACM SIGPLAN Notices 22(12)*, pages 227–242, 1987.