Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Multitudes of Objects: First Implementation and Case Study for Java

Friedrich Steimann[a] Jesper Öqvist[b] Görel Hedin[b]

a. Lehrgebiet Programmiersysteme, Fernuniversität in Hagen, Germany

b. Lund University, Sweden

Abstract In object-oriented programs, the relationship of an object to many objects is usually implemented using indirection through a collection. This is in contrast to a relationship to one object, which is usually implemented directly. However, using collections for relationships to many objects does not only mean that accessing the related objects always requires accessing the collection first, it also presents a lurking maintenance problem that manifests itself when a relationship needs to be changed from to-one to to-many or vice versa. Continuing our prior work on fixing this problem, we show how we have extended the Java 7 programming language with multiplicities, that is, with expressions that evaluate to a number of objects *not* wrapped in a container, and report on the experience we have gathered using these multiplicities in a case study.

ein Vieles, welches kein Eines ist
(*a multitude which is not a one*)
— inspired by Georg Cantor's conception of a set as "jedes Viele, welches sich als Eines denken läßt", i.e., any multitude which can be thought of as a one

1 Introduction

Just like English grammar distinguishes singular and plural, object-oriented programming languages distinguish one object and many objects. However, unlike with English utterances, for which the syntactic difference between the singular and the plural of a noun phrase is usually small, the difference between program fragments dealing with one object and dealing with many objects is often substantial. For instance, while the English utterances "I go to work" and "we go to work" differ only in the pronoun used, in an object-oriented program, the difference would be that between i.goto(work) and for (each : we) each.goto(work), which is cumbersome not only by comparison. The problem, here, is that in object-oriented programming, the multitude denoted by we is reified as a one (usually a collection object), and this one has

Friedrich Steimann, Jesper Öqvist, Görel Hedin. *Multitudes of Objects: First Implementation and Case Study for Java*. Licensed under Attribution 4.0 International (CC BY 4.0). In Journal of Object Technology, vol. 13, no. 5, 2014, pages 1:1–33. doi:10.5381/jot.2014.13.5.a1

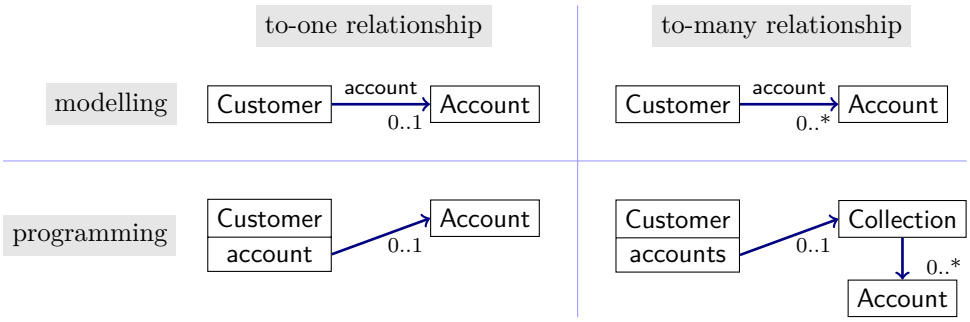


Figure 1 – Relationships to one and to many objects in object-oriented modelling and programming languages: differences

different properties (responds to a different protocol) than the objects it comprises. In particular, the object denoted by we cannot go to work.

Similar to the English language, relational and object-oriented modelling languages make only a small distinction between singular and plural or, more specifically, between one object being associated with one other, or any number of other objects [8, 9, 10, 28, 36]. In these languages, *multiplicity*, also known as *cardinality*, constrains the number of times an object, or entity, may occur in a relationship or association. Hence, a change from singular to plural (or vice versa) requires little more than a corresponding change in multiplicity, as the top half of Figure 1 suggests. By contrast, in object-oriented programming languages multiplicities are commonly coded in the declared type of a variable (which is the type of the related object if it is only one, or the type of a sequence, stream, or collection object if there are more). Here, a change of multiplicity may require a major redesign of the program, as the bottom half of Figure 1 suggests (several more untoward consequences of such a change will be presented below).

In previous work [37], we advocated the introduction of *multiplicities* as annotations of expressions indicating whether an expression is singular or plural, i.e., whether it is expected to evaluate to at most one object, or to any number of objects (not reified). This is to grant the programmer a more uniform treatment of relationships to one object and relationships to many objects in object-oriented programs. In this paper, we present an implementation of our ideas as an extension of the Java programming language using the JastAddJ extensible Java compiler [13], and report on a case study we have conducted.

The remainder of this paper is organized as follows. To motivate our work, we present in Section 2 the peculiarities we observe when implementing multitudes of objects using collections. In Section 3, we briefly describe how enhancing object-oriented programming with multiplicities can generally alleviate the associated problems, with Section 4 specializing our proposal for Java. Section 5 describes our implementation of multiplicities as an extension of the JastAddJ compiler for Java 7. In Section 6, we present qualitative and quantitative findings from a case study extending JUnit with multiplicities. Notes on related and future work conclude.

2 Using Collections for Representing Multitudes of Objects

Undoubtedly, collections are among the most useful abstractions in object-oriented programming: they not only liberate the programmer from manually implementing multitudes of objects as (static) arrays or dynamic data structures (such as linked lists or trees), they also offer a uniform protocol for bulk processing of these object using internal iterators (foreach, select, collect, etc.). And yet, the use of collections for representing many (rather than one) objects comes with a number of peculiarities which make dealing with multitudes of objects very different from dealing with single objects.

2.1 Multiplicity Determines Type

In a program in which every customer can have only a single account, we may see code like

```
Account account;
account = new Account();
account.check();
```

If however a customer can have several accounts, adjustment of just the declaration to reflect this leads to an ill-typed program (faulty expressions underlined):

```
Set<Account> accounts;
accounts = new Account();
accounts.check();
```

Both errors result from the fact that accounts (with a plural "s" appended to express that there can be more than one) now has type Set<Account>, reflecting the changed multiplicity. However, intuitively, what is expressed by the ill-typed program is rather clear: initialize accounts to hold just one account, and then check all accounts (which happens to be only one here). To translate this to standard Java, we would have to write

```
Set<Account> accounts = new HashSet<>();
accounts.add(new Account());
for (Account account : accounts) account.check();
```

which means quite a change to the original program.

2.2 Multiplicity Determines Meaning of `null`

When a variable represents an optional relationship to one object, the value null usually means that there is no relationship (but may also mean failure to initialize):

```
if (account != null) print(account);
else print("no account");
```

For a relationship to many accounts, relating to no account is usually represented using an empty collection:

```
if (accounts != null)
  if (! accounts.isEmpty())
    for (Account account : accounts) account.print();
  else print("no account");
else throw new Error("accounts not initialized");
```

Here, the value null means failure to initialize. Note that having null as an element of a collection makes no sense if the collection is to represent a relationship.

2.3 Multiplicity Determines Subtyping Conditions

If SavingsAccount is a subtype of Account, writing

```
SavingsAccount saving = new SavingsAccount();
Account account = saving;
```

is type-correct. However, when we change to many accounts, the analogue

```
Set<SavingsAccount> savings = new HashSet<>();
Set<Account> accounts = savings;
```

is ill-typed. Instead, we would have to write something like

```
Set<? extends Account> accounts = savings;
```

[24] which does however preclude write access to the set through the variable accounts, greatly limiting its use (especially when considering that the singular account can be used freely).

2.4 Multiplicity Determines Encapsulation Strategy

It is considered good practice in object-oriented programming that the fields of an object are encapsulated and, if necessary, made accessible for clients using setter and getter methods. For collection-valued fields, however, this is different [16]: they are to be updated using add...(...) and remove...(...) methods offered by the encapsulating object (where the ellipses are replaced by the field's name), and if the collection as a whole is to be retrieved, the getter should return a copy or an immutable wrapper [16]. This is so because the collection is considered a representation object which clients should not be able to manipulate directly and of which they should possess no aliases [25]. This brings us directly to the next point.

2.5 Multiplicity Determines Availability of Relationship Aliasing

While assigning an object to a variable with reference semantics always means creating an alias for the object, the semantics differ when the variables are uniformly viewed as implementing relationships to objects, as the following example demonstrates:

```
Account backup = account;
account = null;
if (mistaken) account = backup;
```

Here, backup is an alias for the to-one relationship implemented by account. This is different for

```
Collection<Account> backups = accounts;
accounts.clear();
if (mistaken) accounts = backups;
```

where backups is an alias for the collection denoted, and not for the to-many relationship that is logically established, by accounts. Surely, the problem can be solved by keeping a copy of the collection as backup, but copying is not needed for the to-one case.

2.6 Multiplicity Determines Call Semantics

Continuing the previous example, it may seem awkward that the method

```
void clear(Collection<Account> accounts) {
  accounts.clear();
}
```

performs as intended (i.e., sets the relationship represented by an actual parameter to "no accounts"), while the analogous method for the to-one case

```
void clear(Account account) {
  account = null;
}
```

has no effect on actual parameters. While this may look like a newbie's mistake to the seasoned programmer, it is still indicative of a conceptual chasm, which culminates in the fact that in Java, it is impossible to implement

```
void swap(Object o1, Object o2)
```

with the suggested semantics, while implementing

```
void sort(ArrayList<Object> os)
```

is not a problem. Note that escaping to call-by-reference for swap(..., ...) does not bridge the chasm — not having to do so for collections is just another peculiarity of using them for representing multitudes of objects.

2.7 Multiplicity Determines Meaning of the `final` Modifier

When a variable is declared as final, it means that its value cannot be changed after its initialization. For a variable representing a relationship to a single object this means that the owner of the variable is stuck with the related object for its whole lifetime. For a variable representing a relationship to many objects implemented using a collection, final means that the holder of the relationship is stuck with the collection — its elements, and thus the conceptually related objects, may change freely:

```
final Account forLife = new Account();
forLife = null; // compile error

final Set<Account> allForLife = Arrays.asSet(forLife);
allForLife.clear(); // no problem
```

3 Programming with Multiplicities

The core idea of object-oriented programming with multiplicities as put forward in [37] is that expressions may evaluate directly to any number, or *a multitude*, of objects. This is in contrast to standard object-oriented programming, in which every expression evaluates to either one object or to null and in which multitudes of objects are reified using special container objects (collections, sequences, iterators, etc.). Note that, since multitudes are not reified in our approach, they are always flat, i.e., there is no multitude of multitudes (though it is possible to create a multitude of collections).

Terminological Note We use "multitude of objects" to denote many objects; the term is to be distinguished from "collection of objects" or "set of objects", which each denote an entity in its own right. Note that for this reason it makes no sense to speak of "the elements" or "the members of a multitude", or even of "the objects of a multitude" (since the objects of the multitude *are* the multitude) — if we want to refer to one of many, we say just that, or "one object among a multitude".

3.1 Dynamic and Static Multiplicity

With expressions evaluating to any number of objects, the *dynamic multiplicity* of an expression is defined as the number of objects it evaluates to. In the general case, the dynamic multiplicity of an expression can only be determined at runtime. Therefore, we complement dynamic multiplicity with *static multiplicity*, which can be declared and inferred at compile-time. In the following, the term multiplicity refers to static multiplicity unless stated otherwise.

While dynamic multiplicities are cardinals, we distinguish mainly two (symbolic) static multiplicities, which we call *option* and *any*. *Option* stands for no or one object, while *any* stands for any number of objects. Other static multiplicities are also conceivable (in particular, multiplicity *one*, for precisely one, will be useful; see below); however, since our focus here is on eliminating as much as possible the differences between relating to zero or one and to any number of objects, *option* and *any* suffice.

3.2 Separation of Multiplicity and Type

As long as multitudes of objects are reified, the multiplicity of an expression (i.e., whether it evaluates to one or many objects) is coded in its type: for multiplicity *any*, this type is a collection type (commonly parameterized with the member type, i.e., the type of the elements of the collection), whereas for multiplicity *option*, the type is the type of the optional object (see Section 2.1). Object-oriented programming with multiplicities as put forward in [37] separates multiplicity from type in the declaration of variables and methods: for instance, it allows one to write

```
any Account accounts;
```

instead of

```
Collection<Account> accounts;
```

for declaring that accounts can hold any number of Account objects (note that it cannot hold a collection!), whereas

```
option Account account;
```

which differs only in the multiplicity, is roughly equivalent to

```
Account account;
```

meaning that account can hold either no or one account (see Section 3.4 for the important difference). Note that using multiplicities, both account and accounts have the same type Account; they differ only in their declared multiplicities.

3.3 Assignment Compatibility

While the types of account and accounts are the same and, therefore, do not oppose their mutual assignment compatibility, their multiplicities differ — since *option* is subsumed by *any*, account can be assigned to accounts, and

```
any Account accounts = new Account();
```

is a legal assignment (cf. Section 2.1). An assignment from *any* to *option* is illegal, however; here, a multiplicity downcast (from *any* to *option*) as in

```
account = (option) accounts;
```

is required, but may fail at runtime (namely when accounts holds more than one object).

For variables with multiplicity *any*, assignment is complemented with adding to (+=) and subtracting from (−=) a multitude of objects, where the right-hand side of the update operations can have multiplicity *any* or *option*.

null remains assignment compatible with every reference type; also, it is assignment compatible with both multiplicity *option* and *any* (and means "related to no object" in both cases; cf. Section 2.2).

3.4 Member Access

That account and accounts have the same type means that they respond to the same protocol, i.e., that the same set of methods can be invoked and the same set of fields can be accessed on them. For instance, if class Account defines a method check(), both account.check() and accounts.check() are well-typed; the latter simply means that check() is separately invoked on all objects accounts holds. If Account declares an *option* field bank, accounts.bank returns a multitude of bank objects, namely the banks each account among the multitude of accounts held by accounts is related to. Note that if accounts holds no object, or no account referred to by accounts has a bank associated with it, accounts.bank will evaluate to no object. Since *option* is subsumed by *any*, account.bank will also evaluate to no object if account does not hold an account; note in particular that no null pointer exceptions can arise from dereferencing expressions whose multiplicity is *option* or *any*.[1]

3.5 Aliasing

In object-oriented programming with multiplicities, multitudes of objects are not reified, so multitudes cannot be aliased. This retires the problems noted in Sections 2.3–2.6. In particular,

[1] For the relationship of the multiplicity *option* with the type Option of some functional programming languages (including Scala), see the related work in Section 7

```
any SavingsAccount savings;
any Account accounts = savings;
```

does not cause a covariance problem, since the assignment does not create an alias for a container, but instead assigns accounts the same multitude of objects that savings refers to (by copying pointers just like in the *option* case). It follows that

```
accounts += new Account();
```

does not also add an account to savings (cf. Section 2.3). Likewise, returning accounts as in

```
any Account getAccounts() { return accounts; }
```

does not expose representation to clients (there is no representation object representing the multitude held by accounts) and, in particular,

```
any Account temp = getAccounts();
temp += new Account();
```

does not update the field accounts returned by the getter (Section 2.4). Similarly, after the assignment

```
any Account backups = accounts;
```

(Section 2.5), clearing accounts (by assigning it null; cf. Section 3.3) does not also clear backups, which is therefore still available for restoration. Also, passing a variable into the method clear(...) of Section 2.6, now defined as

```
void clear(any Account accounts) {
  accounts = null;
}
```

does not affect the number of objects that this variable holds, thereby unifying the behaviour for one and many objects. Lastly, the fact that multitudes of objects are not reified unifies the meaning of the final modifier (Section 2.7), which now pertains to variables holding single object and multitudes of objects alike.

3.6 When to and When Not to Use Multiplicities

Our motivation of introducing multiplicities to object-oriented programming is to allow the programmer

- the implementation of relationships (or, more precisely, directed associations [28]) to many objects in a more direct way, and further
- the implementation of relationships to one and to many objects in as much the same way as possible.

This raises the question of what is a relationship, or when multiplicities are to be used.

Experience teaches that programmers will use a construct wherever they deem its use advantageous, so we attempt no dogmatism here. We still make one exception, though: value types, like int, float, boolean (including their wrapper types), or String (whose instances are usually immutable) cannot be the target of a relationship (note that they cannot act as entity types in the entity-relationship model [8]) and

hence cannot be used in combination with *option* or *any* multiplicity annotations. While there are conceptual justifications for this (e.g., people do not relate to their age, an integer value), the main technical reason is that this saves us from defining special semantics for operations on value types (such as +) for operands with *option* or *any* multiplicity (which both include "no object" as a possible value), and also from introducing a ternary logic for handling the case that a boolean expression used in a conditional evaluates to no object. For instance, for a variable declared as option boolean error, it is unclear what if (error) ... means if error has dynamic multiplicity 0. For the same reason, we must exclude that value-typed members are accessed via receiver expressions with *option* or *any* multiplicity, since this can likewise result in dynamic multiplicity 0 (namely when the receiver evaluates to "no object").

4 Multiplicities for Java

While the idea of object-oriented programming with multiplicities as presented in the previous section is language-independent, its adoption in any concrete language invariably requires an individualized integration with existing language constructs. In the following, we present our extension of Java 7 with multiplicities, whose design was driven by our objective to allow a smooth transition between Java programming without and Java programming with multiplicities. Figure 2 has the extended syntax; Figure 3 shows some sample code using it.

```
Modifier ::= ...
 | MultiplicityAnnotation;

MultiplicityAnnotation ::=
   "@any" "(" ReferenceType ")"
 | "@any"
 | "@option"
 | "@bare";

Primary ::=
 | "[[" UnaryExpressionNotPlusMinus "]]"
 | "|[" UnaryExpressionNotPlusMinus "]|";

CastExpression ::= ...
 | MultiplicityCastExpr;

MultiplicityCastExpr ::=
   "(" MultiplicityAnnotation ")" UnaryExpression
 | "(" MultiplicityAnnotation TypeName ")"
   UnaryExpression;
```

Figure 2 – Extension of concrete syntax.

4.1 Multiplicity Annotations

For compatibility with existing Java code, we implemented four static multiplicities:

- *none*, the multiplicity of null (representing no object);
- *bare*, the default multiplicity (and, in particular, the multiplicity of all standard, or legacy, declarations);
- *option*, the multiplicity of entities and expressions representing relationships to no or to one object; and

- *any*, the multiplicity of entities and expressions representing relationships to any number of objects.

Multiplicities determine assignment compatibility according to the order

$$none < bare < option < any$$

i.e., every multiplicity is assignment compatible with itself and all greater ones.

Hidden Collections To give the programmer control over the nature of multitudes, and also for interfacing with legacy Java code that uses collections (see below), *any* multiplicities may be parameterized with a collection type C whose definition has a single type parameter (e.g., List<E>). This type will be used to instantiate a *hidden collection* holding the multitude of objects. To acknowledge the widespread use of abstract collection types in Java programs, C may be an abstract class or an interface.

Syntactic Integration The multiplicity *none* does not occur in program texts; the other multiplicities appear as annotations @bare, @option, and @any, respectively (see Figure 2). Since *bare* is the default multiplicity, it occurs only in multiplicity downcasts from *option* or *any* to *bare* (see below).

original (without using multiplicities)

```
class Subject implements Observable {
  Set<Observer> obs = new HashSet<>();

  public void addObserver(Observer o) {
    if (o == null)
      throw new NullPointerException();
    obs.add(o);
  }

  public void deleteObserver(Observer o) {
    obs.remove(o);
  }

  public void notifyObservers(Object arg) {
    for (Observer o : obs)
      o.update(this, arg);
  }

  public void deleteObservers() {
    obs.clear();
  }

  public int countObservers() {
    return obs.size();
  }
}
```

using multiplicities

```
class Subject implements Observable {
  @any(HashSet) Observer obs;

  public void addObserver(@option Observer o) {

    obs += o;
  }

  public void deleteObserver(@option Observer o) {
    obs -= o;
  }

  public void notifyObservers(Object arg) {

    obs.update(this, arg);
  }

  public void deleteObservers() {
    obs = null;
  }

  public int countObservers() {
    return |[obs]|;
  }
}
```

Figure 3 – Example of implementing class Subject of the Observer Pattern, with and without using multiplicities (differences highlighted)

4.2 Declarations

The multiplicity annotations @option and @any may appear (in the position of modifiers; cf. Figure 2) in all declarations of reference-typed entities, except where the type is a wrapper type (such as Boolean) or String (see Section 3.6 for the reasons). If an entity is annotated with @any(C) and C is a concrete collection class, the compiler uses C as the class of the hidden collection that holds the multitude of objects the annotated entity denotes; if C is abstract or not provided, the compiler automatically picks a suitable concrete class (note that the single type parameter of C is instantiated with the type of the declaration). Thus, multitudes of objects *are* implemented using collections; however, this implementation is strictly under the hood and, in particular, these collections cannot be accessed from the program as objects (they cannot be aliased). Entities annotated with @option are not implemented using collections; however, the compiler gives their value null a special meaning (see below).

Final Declarations A declaration final @option T v means that, after its initialization, the value of v cannot change (i.e., v always refers to the same object). That v is not assigned new values after initialization is ensured (by the compiler) as usual. A declaration final @any T v likewise means that v cannot be updated after its initialization (i.e., it always refers to the same objects), where updating includes assignment (=), adding objects (+=), and removing objects (−=); this is ensured by the very same means. Hence, no immutable collections are required to express the immutability of a multitude.

4.3 Interfacing with Collections: Wrapping and Unwrapping

To interface multiplicity *any* with code that uses (bare) collections for representing multitudes of objects, we must be able to wrap a multitude in a collection object, and to unwrap it from a collection object. We use double square brackets ("[[. . .]]") for both purposes (see Figure 2). If the argument expression has static multiplicity *any(C)*, the result is a (fresh) collection of type C holding the objects the expression evaluates to; if it has multiplicity *option*, the result is a new instance of class ArrayList which either holds the object the expression evaluates to, or is empty if it evaluates to null. We call this *wrapping*. If the argument expression has static multiplicity *bare* and is a collection, the result is the multitude of objects that the collection holds (which is internally represented using a fresh collection of the same type). We call this *unwrapping*. Unwrapping is particularly useful for initializing final variables with multiplicity *any*:

```
final @any Account accounts =
  [[Arrays.asList(new Account(), new Account())]];
```

The expression [[null]] is not allowed.

4.4 Number of Objects

The dynamic multiplicity, or the number of objects an expression evaluates to, is computed using "|[. . .]|" (see Figure 2). In case the argument expression has multiplicity *any*, it returns the size of the underlying collection; if the expression has multiplicity *option*, it returns 0 if it evaluates to null, and 1 else.

4.5 Casts

As for types, multiplicity upcasts (e.g., from *option* to *any*) are always safe and therefore may remain implicit. Multiplicity downcasts from *any* to *option* or *bare*, however, may fail, namely when the *any* expression being cast evaluates to more than one object. Therefore, the compiler inserts a runtime multiplicity check for all such casts which, upon failure, throws a multiplicity cast exception. Casts from *option* to *bare* are also always safe; since unlike for *option* receivers, accessing members on *bare* receivers can lead to null pointer exceptions, we will require explicit downcasts from *option* to *bare* (see below for examples of where this is needed).

4.6 Expressions

With new syntax given meaning as above, we now turn to the impact multiplicities have on standard Java expressions.

Update Assignment (=) makes the variable on the left-hand side refer to the objects the right-hand side refers to. Given the arbitrariness of the definition of "identity of multitudes of objects" for multiplicity *any* (see below), we defined assignment pragmatically: the hidden collection holding the multitude of the left-hand side is first emptied (cleared), and then all objects of the hidden collection representing the right-hand side are copied into it using its addAll(...) method. If the right-hand side has multiplicity *option*, its object (if any) is added to the collection using add(...). If the left-hand side has multiplicity *option*, the object that the right-hand side evaluates to is assigned to it.

Adding (+=) is only allowed for left-hand sides with *any* multiplicity and adds the object(s) of the right-hand side (if any) to it, again using add(...) or addAll(...). Removing (−=) works accordingly, using the corresponding remove methods. Note that the programmer can override the meaning of += and −= by supplying her own collection implementations to the *any* annotations in the declarations.

Member Access Accessing a member m on a receiver r with multiplicity *any* unfolds to accessing m on every object among the multitude r evaluates to, in the order provided by the iterator of the hidden collection holding the multitude. If m is a field or a non-void method, r.m evaluates to a multitude of objects, independently of whether m has multiplicity *option* or *any* (recall that multiplicities are always flat). As argued in Section 3.6, m must not be *bare*; if it is, the receiver must be cast to *bare* first (cf. Section 4.5 and 6.2.2).

Since *option* is subsumed by *any*, accessing m on r having multiplicity *option* behaves exactly as if r had static multiplicity *any* and dynamic multiplicity 0 or 1. In particular, if r is null, evaluating r.m does not raise a null pointer exception — it simply evaluates to null (for "no object"). However, deviating from receiver multiplicity *any*, r.m has multiplicity *option* for *option* members (see Table 1).

If m is a method, parameter passing works according to the rules of assignment (see under "Update" above). In particular, the formal parameters do not receive aliases to the hidden collections holding the objects of formal parameters having multiplicity *any*. Similarly, m does not return

r \ m	*bare*	*option*	*any*
bare	*bare*	*option*	*any*
option	N/A	*option*	*any*
any	N/A	*any*	*any*

Table 1 – Multiplicity of member access expressions `r.m`

aliases to the hidden collection representing the returned expression. Note that, with respect to multiplicity, overriding methods must be contravariant in the formal parameter multiplicities (i.e., *option* can be overridden with *any* etc.) and covariant in the return multiplicities (i.e., *any* can be overridden with *option* etc.). Figure 4 has an example of a covariantly overridden method (getLeaves()).

Test for Identity Strictly speaking, a test for identity ("==") does not make sense for multitudes of objects: if multitudes are not reified, how can they be identical? On the other hand, if the dynamic multiplicities of the left-hand side and the right-hand side of such a test are 0 or 1, there seems little choice in defining the meaning of ==: it is true if and only if either both evaluate to the same object, or both evaluate to null. For greater numbers of objects, it would seem reasonable to require that both sides have the same dynamic multiplicities; yet, this means that even immediately after an assignment of an expression having multiplicity @any(List) to a variable having multiplicity @any(Set), identity may not be given (due to the dropping of duplicate objects). In practice, what it means for two multitudes to be identical (or only equal) is at least as variable as what it means for two collections to be equal, so that eventually, the programmer must be given control over this question (by letting her implement her own tests). Therefore, we made an arbitrary choice for == and implemented it as each object from each multitude occurring exactly the same number of times in both multitudes. Note that for a test of equality using the equals(...) methods provided for collections, the multitudes must be wrapped first (see Section 4.3).

Iteration over Multitudes While member access on a multitude results in an implicit (hidden) iteration over its objects (see "Member Access" above), there are iterations that require explicit access to the individual objects of the multitude, for instance to apply a filter, because there are case analyses to be made, or because the objects are to be used as arguments to operations or method calls (see Section 6.1.1 for examples). In these cases, wrapping a multitude in a collection (see Section 4.3) allows

original (without using multiplicities)

```
abstract class Composite {
  abstract List<Leaf> getLeaves();
}

class Component extends Composite {
  List<Composite> children = new ArrayList<>();
  List<Leaf> getLeaves() {
    List<Leaf leaves = new ArrayList<>();
    for (Composite child : children)
      leaves.addAll(child.getLeaves());
    return leaves;
  }
}

class Leaf extends Composite {
  List<Leaf> getLeaves() {
    return Arrays.asList(this);
  }
}
```

using multiplicities

```
abstract class Composite {
  abstract @any Leaf getLeaves();
}

class Component extends Composite {
  @any Composite children;
  @any Leaf getLeaves() {


    return children.getLeaves();
  }
}

class Leaf extends Composite {
  @option Leaf getLeaves() {
    return this;
  }
}
```

Figure 4 – Example of implementing a composite structure with and without using multiplicities (differences highlighted)

us to iterate over its objects as usual, i.e., using for, while, and do. For the special (and presumably most frequent) case of using the enhanced for loop, multitudes can be used in place of an iterable object without prior wrapping in a collection.
E.g., we can write

```
for (Account account : accounts) ...
```

if accounts is declared as @any Account accounts. Note that, if the type of accounts was a subtype of Iterable, the for-loop would still iterate over the multitude, and not the elements of the iterable(s). This is also true if the (declared) multiplicity of accounts is @option (in which case the for-loop behaves more like an if-statement).

While the Java 8 Stream API adds another abstraction over collections which makes them more convenient to use by removing the need for external iteration in many cases, a stream is just another container — and hence another reification — of a multitude of objects. However, using the wrapping mechanism (see above and Section 4.3), the full repertoire of stream operations can be invoked on multitudes; in case of the above accounts example, one simply needs to write [[accounts]].stream()....

5 Implementation

We implemented multiplicities for Java as described above as an extension to JastAddJ [13], an extensible Java compiler implemented using reference attribute grammars [12, 20], and which currently supports Java 7 [30]. The extension comprises 44 source lines of JastAdd code for the syntax, 672 lines for the static semantics, and 1,180 lines for code generation. The multiplicity compiler can be downloaded from `https://bitbucket.org/joqvist/multiplicities`.

Abstract Syntax Abstract syntax is added to support multiplicity modifiers and expressions for wrapping/unwrapping, cardinality, and multiplicity casts, as shown in Figure 5. Each rule corresponds to a class representing an abstract syntax tree (AST) node, extending and reusing existing classes in the JastAddJ compiler, like Modifier, Access, and Expr. Much of the static semantics behaviour, like name analysis, is reused as is from JastAddJ, but type analysis is refined in the extension, supplying new attribute grammar equations that define appropriate attribute values to handle multiplicities.

```
abstract MultiplicityModifier extends Modifier;
AnyModifier extends MultiplicityModifier ::=
  ContainerType:Access;
AnyDefaultModifier extends MultiplicityModifier;
OptionModifier extends MultiplicityModifier;
BareModifier extends MultiplicityModifier;
MultiplicityWrap extends Expr ::= Expr;
MultiplicityCardinality extends Expr ::= Expr;
MultiplicityCast extends Expr ::=
  Modifier:MultiplicityModifier
  [TypeAccess:Access]
  Expr;
```

Figure 5 – Abstract syntax of extension with multiplicities

Type Analysis In JastAddJ, each type is represented by a unique AST node. Type checking, as used in assignment, parameter passing, etc., relies on the binary property

of assignment compatibility which is implemented by comparing two type nodes, using double dispatch to encode the type lattice in an extensible way [13]. To handle types with multiplicities (other than *bare*), we construct synthetic multiplicity nodes that decorate ordinary type nodes. This allows us to compare different multiplicities with each other and with *bare* (non-decorated) types, again using the double dispatch pattern.

The synthetic nodes are constructed using the attribute grammar mechanism of *non-terminal attributes* (NTAs) [40], i.e., attributes whose values are new AST children. In JastAddJ, all attributes are computed automatically by the attribute grammar evaluator, and on demand, constructing only the synthetic decorating nodes that are needed for a particular program.

As an example, consider the following code fragment:

```
@any Account accounts;
...
accounts += new Account();
```

Figure 6 shows parts of the corresponding attributed AST. While the new expression is bound to the (*bare*) Account type, the declaration and access of accounts are bound to the AnyMult node that decorates the Account type.

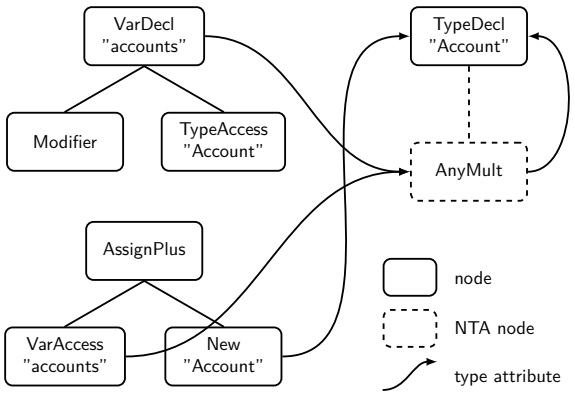


Figure 6 – AST with reference attributes (see text)

Member Access Multiplicities affect the analysis of member accesses (qualified expressions). In regular Java code, the type of any qualified expression is the type of the object on the right-hand side of the rightmost dot. The qualifiers are only used for looking up the declaration of the rightmost part. However, with multiplicities it is not sufficient to only look at the multiplicity of the rightmost part – the qualifying expression multiplicities may affect the multiplicity of the whole expression. For example, as discussed in Section 3.4, if the Account class has a field @option Bank bank, the expression accounts.bank will have the multiplicity *any*, although bank has the multiplicity *option*. This is handled by extending the type analysis with attributes to find the multiplicity of the left-hand part of a dot expression. The multiplicity of the entire dot expression is computed using both the multiplicity of the left and right parts, following Table 1.

Code Generation The code generation constitutes the bulk of the multiplicities implementation, translating from the higher-level operations on multitudes to cor-

responding lower-level for-loops, and handling all the different combinations of multiplicities in assignments and expressions. The implementation largely follows the translation scheme to a multiplicity-free program proposed in [37]; it is not repeated here for space reasons (Section 4 provided an outline, however).

Copying Hidden Collections In the bytecode, multiplicity *any* is represented by a collection object (the hidden collection), and care must be taken to not create aliases of these objects when multiplicity values are copied. For this reason, we create a copy of the collection object

- when passing an *any* as an argument to a method or constructor,
- when returning an *any* from a method,
- at an explicit (@any) cast, and
- when the multiplicity wrap expression either wraps or unwraps an *any*.

The assignment to an *any* does not need to create a copy of the hidden collection of the right-hand side — instead, the objects of this collection are added to the cleared hidden collection of the left-hand side.

Copying of collections can in many cases be avoided through static analysis and/or lazy copying: In certain cases, we can deduce statically that the original collection cannot be used anymore, for instance when returning the value of a local variable. In these cases, copying is not needed. Another optimization strategy is to represent a multitude by a wrapper object that contains an internal collection. Copying can then be implemented lazily by doing a shallow copy of the wrapper object, delaying the copying of the internal object until it is modified. By letting the wrapper keep a reference count, copying of the internal collection can be avoided for wrappers whose internal collection is not shared by other wrappers. We implemented such lazy collections as a library that we then benchmarked against the regular, non-lazy collections in our case study (see Section 6.3 for the performance discussion).

6 Case Study

To assess the impact of using multiplicities in a representative case study, we looked for a subject program

1. that uses a wide array of accepted object-oriented coding idioms so that multiplicities can be evaluated in a spectrum of constructions typically encountered in object-oriented programming,
2. that is tightly covered by test cases so that accidental changes of functional behaviour induced by the use of multiplicities would quickly be discovered, and
3. that can be run using file-based input that is openly available for reproduction of our performance observations (replication).

Given these criteria, we selected the widely known regression testing framework JUnit 4.0

1. since it is renowned for its consistent use of design patterns and, generally, for its exemplary object-oriented style,

2. since its own test suite is comprehensive, and
3. since open-source programs are available whose test suites execute it, giving us the standardized program runs we wanted.

Note in particular that every JUnit test suite tests not only the program under test, but also JUnit itself against the suite's oracle: all and only the tests of a program that pass using the original version of JUnit should also pass using our modified version using multiplicities. We selected JUnit 4.0 rather than one of its successors since it is manageable in size and since it contains fewer features that are not used by the majority of available JUnit test suites.

To manually obtain a version of JUnit that utilizes multiplicities in a way that is both reproducible and that we deem to be representative of how multiplicities will be used in practice, we changed the multiplicity of every field having a collection type whose element type (type parameter) is not a value type to `@any` (and removed the collection type from the type declaration), and every other field not having a value type to `@option`. We then changed the multiplicity of every other variable and method return as required by the assignment compatibility and overriding rules of multiplicities (see Section 4.6), unless where use of APIs (method invocation and subclassing) required *bare* parameters (in which case a cast to *bare* was introduced). The results of this procedure are summarized in Table 2.

	any	*option*	*bare*	total	value-typed[†]
fields	11	44	121	176	29
returns	9	34	249	292	108
formals	3	83	642	728	403
locals	7	34	482	523	380
casts to	2[§]	21[$]	92	115	
total	32	216	1586	1834	920

[†] included in *bare*
[§] implicit upcasts
[$] all downcasts from *any*

Table 2 – Multiplicities in declarations and casts

6.1 How Introduction of `@any` Changes Program Source

There were 11 collection-typed fields whose type parameter (element type) was not a value type (cf. Section 3.6), and which we changed to multiplicity *any*. Of these, 5 were originally declared `final`; in 4 of these cases, the `final` modifier had to be dropped since the multitudes were actually modified after initialization (cf. Section 2.7). Introducing the multiplicity `@any` (and changing the types; cf. Section 3.2) for these 11 fields required the subsequent change of 3 formal parameters (one of which involved the removal of a wildcard; cf. Section 2.3), of 9 method returns, and of 7 local variables (giving us a total of 30 introduced `@any` annotations; see Table 2). Of the remaining 27 uses of collections, 9 were required by APIs, 5 held instances of value types, 3 were required by concurrent modification (iter/remove; see Section 6.1.2), and the rest was used by local variables that never got a field assigned to it (they could also have been changed to multiplicity *any*).

Together with the introduction of @any annotations, we replaced 12 invocations of add(...) and 1 of addAll(...) with +=, and 2 invocations of remove(...) with −=; at the same time, 10 invocations of size() where replace with |[...]| and 5 invocations of isEmpty() were replaced with a test for null (cf. Section 3.2). There were 27 indexed accesses to list elements (using get(...)) in the original program where the list was replaced by an *any* multitude; all but 5 of these could be removed using multiplicity downcasting (see Section 6.1.3); the remaining 5 required wrapping (see Section 6.1.2).

6.1.1 Loop Elimination

One of the supposed benefits of introducing *any* multiplicity is the elimination of loops (see Sections 2.1 and 3.4). And indeed, 5 for-loops over the elements of collections could be replaced by plain member access on a corresponding multitude of objects. For instance, we replaced the loop

```
for (Runner each : fRunners)
  each.run(notifier);
```

(from CompositeRunner.run(RunNotifier)) with

```
fRunners.run(notifier);
```

In addition, even where the iteration variable is not used as the left-most receiver in the loop expression (as above), it may still be possible to eliminate the loop. For instance,

```
for (Runner runner : fRunners)
  spec.addChild(runner.getDescription());
```

(from CompositeRunner.getDescription()) was rewritten to

```
spec.addChild(fRunners.getDescription());
```

after the multiplicity of the formal parameter description in

```
public void addChild(Description description) {
  fChildren += description;
}
```

had been changed to @any (note how this does not affect the implementation of addChild(...)).[2] However, because *bare* members may not be accessed on *option* or *any* receivers (Section 4.6), this required the declaration of @option as the returned multiplicity of getDescription(...) which, since *option* is not assignment compatible with *bare* (Section 4.1), required the subsequent introduction of 32 more @option annotations throughout the program. Yet, given that @any and @option annotations are designed to be used together, this does not appear to be counterproductive.

With a little redesign of programs, loop elimination can be pushed even further. For instance, we found (in method createTest(...) from class JUnit4TestAdapterCache) the loop

```
for (Description child : description.getChildren())
  suite.addTest(asTest(child));
```

[2] In fact, increasing parameter multiplicity of add...(@option) methods like the above allowed us to drop two addAll...(@any) methods from the JUnit 4.0 source (they are now subsumed by the add...(@any) methods).

Here, the loop variable child is the argument of another method so that introducing multiplicity @any for the parameter of addTest(...) as above is not sufficient — asTest(...) would need to be changed to accept and return *any* multiplicity as well, which would require a major reworking of its implementation. However, as it turned out, asTest(...) can straightforwardly be moved to class Description (the class of its formal parameter, using the refactoring Move Method [16]), so that the loop can be replaced by

```
suite.addTest(description.getChildren().asTest(this));
```

which is not only more succinct, but also more fluent[3] than the original phrasing. In fact, this minor refactoring even allowed us to remove a loop that was designed to fill a collection: we turned

```
List<Test> returnThis = new ArrayList<Test>();
for (Description child : description.getChildren())
  returnThis.add(child.asTest(this));
return returnThis;
```

(from JUnit4TestAdapterCache.asTestList(...)) into

```
return new ArrayList<Test>(
  [[description.getChildren().asTest(this)]]
);
```

in which the *any* multitude returned by asTest(...) is wrapped in a collection (note that API calls explicitly expect the collection here; hence the name of the method, "asTestList"!).

Of the 11 loops on the elements from a multitude that could not be removed, 2 contained accesses of value-typed members (methods for counting the number of leaves in a composite structure; see Figure 4 for how this can be simplified using collections), 3 used explicit iterators for removing elements from the originally underlying collection (cf. Section 6.1.2), and 6 had complex loop bodies that would have required major refactorings to cast them to member access on *any* expressions.

6.1.2 Wrapping and Unwrapping Multitudes

We required a total of 15 wrappings or unwrappings:

- In 5 cases, wrapping a multitude in a list was necessary because indexed access to individual objects was required and the dynamic multiplicity was not known to be 0 or 1 (in which case a downcast to *option* would have been sufficient; cf. Section 4.5).
- In 3 cases, wrapping a multitude into and subsequently unwrapping it from a local list-typed variable was necessary because of iter.remove() loops.
- The remaining 4 wrappings and unwrappings were due to API calls.

[3] "fluent" in the sense of a "fluent API": see http://www.martinfowler.com/bliki/FluentInterface.html

6.1.3 Multiplicity Casting

In Java without multiplicities, a multiplicity upcast (from *option* to *any*) requires the wrapping of a single object in a collection. For instance, the method whose header is declared as

```
List<Throwable> getCauses(Throwable cause)
```

(from class ErrorReportingRequest) returns the expression Arrays.asList(cause) as a special case. In case cause was null, it would need to return an empty list, involving yet another clumsy construction (see Section 6.1.5). Using multiplicities, the same method is declared as

```
@any(List) Throwable getCauses(@option Throwable cause)
```

for which cause is a type-correct and multiplicity-correct return expression (the multiplicity upcast is implicit here).

Multiplicity downcasts (from *any* to *option*) are somewhat more involved. In standard Java, this would require the test of the size of a collection and, in case it is 1, the extraction of the sole element of the collection (the cast would result in null if size is 0, or else raise an exception). Indeed, we found 24 of constructions such as

```
Failure failure= result.getFailures().get(0);
assertEquals(expected, failure.getDescription());
```

in JUnit, which silently assumes that there is at least one failure and ignores possible failures beyond the first (actually, it leaves unstated whether there may be additional failures). Using multiplicities, we can rewrite the first line to

```
@option Failure failure= (@option) result.getFailures();
```

which makes the cast explicit and states that there should be at most one failure (which proved to be the correct assumption in 17 out of the 24 occurrences of this pattern).

6.1.4 Enforce Proper Encapsulation of Multitudes

As noted in Section 2.4, fields holding collections should be encapsulated and not be passed to clients via getters. Nevertheless, we find in JUnit's class Description the method

```
public ArrayList<Description> getChildren() {
  return fChildren;
}
```

allowing clients to bypass the public method addChild(...) supplied by the same class for directly manipulating the children of Description objects. After replacing the declaration of the field fChildren in Description with

```
@any(ArrayList) Description fChildren;
```

and adjusting the above declaration of getChildren() accordingly, an invocation of getChildren().add(...) will have no effect on fChildren, since getChildren() no longer returns an alias of it (Section 3.5). As it turns out, however, the sole occurrence of a manipulation of fChildren in JUnit via getChildren() is in the body of addChild(...) itself:

```
public void addChild(Description description) {
  getChildren().add(description);
}
```

Here, the idea of Self Encapsulate Field [16] clearly conflicts with how collections should be encapsulated (see Section 2.4). Using multiplicities, the body of addChild(...) is rewritten to

```
fChildren += description;
```

while that of getChildren() can remain as is, without granting true clients access to fChildren.

While the use of multiplicities enforces proper encapsulation as shown above, it can also help avoid explicit cloning, as found in class TestResult:

```
synchronized List<TestListener> cloneListeners() {
  List<TestListener> result= new ArrayList<>();
  result.addAll(fListeners);
  return result;
}
```

Here, using multiplicity *any* it suffices to return fListeners in the body of the method (which needs to remain synchronized — all multiplicity operations are non-synchronized by default).

6.1.5 Uniform Use of `null`

The fact that relating to no object in a to-many relationship is commonly represented by an empty collection (cf. Section 2.2) has led to the introduction of special collection classes (e.g., Collections.EmptyList, Collections.EmptySet, both from java.util) whose sole instances represent an empty collection. For instance, the method declared as

```
List<Throwable> validateAllMethods(Class<?> clazz)
```

(from class ParameterizedTestMethodTest) returns Collections.emptyList() as a special case (an upcast from multiplicity *none* to *any*). Replacing the declaration of the method with

```
@any(List) Throwable validateAllMethods(Class<?> clazz)
```

allows the method to return null instead, which has the same meaning as null for @option, i.e., is subsequently interpreted as no object (and, unless it is cast to @bare, cannot cause a null pointer exception). Note that the fact that, unlike Collections .emptyList(), the returned multitude is mutable maintains behavioural subtyping [22]: while

```
@any(List) Throwable result = null;
result += new Throwable("it's OK!");
```

is indeed OK, the seeming equivalent

```
List<Throwable> result = Collections.emptyList();
result.add(new Throwable("not OK!!"));
```

causes an "unsupported operation" exception.

6.1.6 Uniform Call Semantics

The fact that method calls are by value effectively (i.e., a method cannot modify the multitude that it gets passed; cf. Section 3.5) means that methods such as void Collections.sort(List<T>) (which would need to be rewritten to void Collections .sort(@any(List) T)) no longer work, simply since sorting has no effect on the multitude that is passed into the method. To fix this, we wrote our own sort method that returns a sorted multitude which can be assigned back to the variable holding the original multitude. Specifically, we changed 2 invocations of the kind

```
Collections.sort(fRunners, ...)
```

(here from class CompositeRunner) to

```
fRunners = Multiplicities.sort(fRunners, ...)
```

where Multiplicities is a helper class analogous to Collections. Note how this makes clear why fRunners cannot be declared final, since the multitude is in fact changed (even though the collection secretly holding it has remained the same object; cf. Section 2.7).

6.2 How Introduction of `@option` Changes Program Source

Changing the remaining fields that did not have value types to multiplicity *option*, and subsequently also formal parameters, method returns, and local variables as required by the rules of Section 4.1, gave us a total of 44 fields, 34 returns, 83 formal parameters, and 34 locals, all with multiplicity *option* (see Table 2).

6.2.1 Elimination of Tests for Not Null

Just like the use of *any* can eliminate loops, the use of *option* can eliminate tests for not null (Section 4.6). As it turns out, however, JUnit does not make much use of the value null representing "no object": in fact, in the whole of JUnit there is no test for not null on a field that could be declared with @option, and only a single test for null (which is however only used for lazy initialization of the field). However, there are some tests for not null on local variables, one of which,

```
Runner childRunner= Request.aClass(each).getRunner();
if (childRunner != null)
  runner.add(childRunner);
```

(from method ClassesRequest.getRunner()), we could rewrite to

```
runner.add(Request.aClass(each).getRunner());
```

This was possible since method add(...) accepts *any* multiplicity and getRunner() returns *option* multiplicity, and since null uniformly means "no object" for *option* and *any* multiplicities (see Sections 3.2 and 4.6).

6.2.2 Multiplicity Casting

While explicit and implicit multiplicity casts to *option* and *any* avoid clumsy coding idioms (Section 6.1.3), the current well-formedness rules of multiplicities may also require explicit downcasts to *bare* (cf. Section 4.5), which can be a nuisance. Specifically, the fact that value-typed members (which must be *bare*) may not be accessed on receiver expressions with multiplicity *option* (Sections 3.6 and 4.6) can require annoying casts. For instance, in

```
public int countTestCases() {
  return ((@bare) fRunner).testCount();
}
```

(from class JUnit4TestAdapter) the cast (@bare) is required since fRunner has multiplicity *option* (meaning that it may evaluate to no object) and testCount() returns an integer. Even though the cast (@bare) can be read as a warning that a null pointer exception may occur here (which can never occur when dereferencing *option* or *any* receivers; see Section 3.4), given that we needed to insert 62 such casts in JUnit (cf. Table 2; the remaining 30 casts to *bare* were needed for interfacing the JDK and assertions), not all programmers will regard this aspect of our language design as ideal. An elegant solution to this problem seems to be the introduction of *one* as an additional multiplicity annotation (for relating to precisely 1 object) and to allow access of *bare* members on *one* receivers with resulting multiplicity *bare*. However, since this would require our notion of multiplicities to be integrated with existing not-null annotations and checks, we have left this to future work (Section 8.1).

6.3 Performance Observations

To check the correctness of our multiplicity compiler, we ran JUnit's own test suite on our multiplicity-enhanced version of JUnit (named "JUnit-M"), and also on the test suites of three additional multiplicity-free benchmark programs listed in Table 3. All tests gave the same results, suggesting that the modified and the original version of JUnit are functionally equivalent. There were 25 test cases that failed in both versions (for AC Lang), because they require a newer runtime version of JUnit. We decided to keep these tests since they exercise failing behaviour in JUnit. All other tests that could be compiled with JUnit 4.0 (cf. Table 3) passed.

To check how multiplicities affected the execution time of JUnit-M, we compared running the following different compiled versions of JUnit:

- *ju4jc*: original JUnit 4.0 compiled using javac from OpenJDK 7, i.e., the reference compiler for Java.
- *ju4jj*: original JUnit 4.0 compiled using JastAddJ for Java 7.
- *ju4m*: JUnit-M 4.0 compiled with our multiplicity-enhanced compiler
- *ju4l*: JUnit-M 4.0 compiled with a variant of our multiplicity-enhanced compiler that uses lazy copying of collections, as described at the end of Section 5.

We used these four different versions of compiled JUnit to run the test suites in Table 3, all of which were compiled using javac.

Steady-state performance We measured execution time in steady state, i.e., after running for a while so that the optimizing JIT compiler has *warmed up*, and reached a stable state. This is a relevant test scenario for long running applications. However, to use this method on the JUnit test suite, we had to remove two test cases that include an infinite loop that can be stopped only by a call to System.exit() (thereby terminating the host JVM).

To measure on steady state, we used the *multi-iteration determinism* method for benchmarking from Blackburn et al. [5], which includes the following steps:

Subject program and Version	*Number of Test Cases*
Apache Commons Codec 1.3	191
Apache Commons Lang 3.0	1923[†§]
Jaxen 1.1.6	716[$]
JUnit 4.0	255

[†] excluding 10 that we had to remove because they could not be compiled with JUnit 4.0
[§] 25 of these tests fail both with and without multiplicities because they should normally be run with JUnit 4.7 (see text)
[$] excluding 2 that were removed because they contain an infinite loop (see text)

Table 3 – Subject programs used in the evaluation.

1. The benchmark is iterated $N - 1$ times in the same JVM to achieve steady-state for the JIT.
2. JIT optimization is then turned off to not further affect the measurements.
3. One more iteration of the benchmark is made, but is not measured.
4. Finally, K iterations are made, measuring the execution time of each.

During different runs (consisting of $N + K$ iterations of the benchmark), the JIT may stabilize on different states, due to the non-determinism of the JIT optimization. For this reason, we make R runs for each benchmark, and compute the arithmetic mean of the means of the K iterations in each run, and the 95% confidence interval, as suggested by Georges et al [17]. For our measurements we chose $R = 15$, $N = 30$, and $K = 20$. Table 4 shows the results of our steady-state experiments.

	ju4jc	ju4jj	ju4m	ju4l	ju4m/ju4jj loss
AC Codec					
mean	172	169	178	204	0.0530
conf. int.	±16.5	±15.2	±15.5	±2.7	
AC Lang					
mean	6747	6748	6750	6758	0.0003
conf. int.	±2.6	±2.6	±2.2	±2.3	
Jaxen					
mean	249	248	249	258	0.0033
conf. int.	±3.1	±1.0	±1.3	±1.3	
JUnit					
mean	845	848	853	867	0.0053
conf. int.	±2.2	±2.3	±2.6	±3.1	

Table 4 – Execution times (in msecs).

In comparing *ju4jj* and *ju4m*, we anticipated there to be a performance loss due to copied collections and extra null checks. We can see that there is a tendency to a slight performance loss when using multiplicities for all four benchmarks. However, the confidence intervals overlap, and the difference between the means is only 5%

for AC Codec, and less than 1% for the other benchmarks. We therefore regard the performance loss as negligible. We can also note that the performance of the javac compiler and the JastAddJ compiler (*ju4jc* and *ju4jj*) are almost the same, indicating that the results should transfer to javac, should one wish to implement multiplicities there.

Using Lazy Copying Table 4 also shows the results from running *ju4l*, i.e., JUnit-M compiled with a variant of our multiplicity-enhanced compiler that implements lazy copying of collections, as discussed at the end of Section 5. Unfortunately, the results show that the use of lazy copying degrades the performance, rather than improving it. We measured the number of copied collections (Table 5) and their sizes, and found that around half of the collections had size 0 and that less than 1% had a size larger than 10. Apparently, because the collections are so small, the cost of copying the hidden collection is lower than the cost of delegating all method calls through an intermediate lazy collection. Further investigation is needed to see if the implementation of the lazy copying can be improved, and if it can be useful for other benchmarks.

	AC Codec	*AC Lang*	*Jaxen*	*JUnit*
collection copies	1007	20326	8038	8105
avoided using lazy	778	16516	6554	6884
checking not null	1045	8417	3112	3310

Table 5 – Instruction overhead

6.4 Discussion

As the examples of Figure 3 and Figure 4 suggest, savings in terms of the number of tokens used in a program fragment can be considerable. Also, Section 6.1.1 presented several interesting examples of loop elimination enabled by member access on multitudes of objects. In a complete program, however, savings are diluted, and the total number of tokens can even increase because of the additional annotations required in declarations. In fact, in our case study the multiplicity-enhanced version has 151 more tokens than its original. However, this increase is explained by the additional annotations used in declarations, whereas the number of tokens in the other statements (instructions) are reduced. The possible reduction of tokens in the instructions is currently diminished by the casts to *bare* that we had to introduce for interfacing with API code and accessing *bare* members (Section 6.2.2). We expect these numbers to improve with the introduction of *one* as an additional multiplicity, and of course with the migration of APIs.

7 Related Work

Smalltalk not only comes with a powerful collections library, with its indexed instance variables it also offers a way of directly associating one object with a multitude of other objects, without reifying this association [19]. However, since indexed integer variables are unnamed (they are similar to the so-called indexers of the .NET languages [23]), there can be only one set of indexed instance variables per object, limiting their use for implementing relationships (of which an object may have many). And yet, indexed

instance variables share with our *any* fields that two objects cannot share the same set of indexed instance variables (i.e., there is no aliasing of multitudes).

The object constraint language (OCL) [6, 27], which is used to express conditions of well-formedness of UML models, allows the dereferencing ("navigation") of attributes and associations with arbitrary multiplicities using the dot notation. However, OCL still reifies multitudes of objects using collections; the difference between one and many objects (singular and plural) is mitigated only slightly by allowing collection operations to be applied to single objects also. This is different for Alloy [21], a textual modelling language which maps object-orientation to relational logic and in which the notion of multiplicity is also prominent. Unlike OCL, Alloy does not distinguish between scalars and sets, and treats scalars as singletons. This largely removes the differences between one and many objects from Alloy expressions (which we strive for also); however, like OCL, Alloy is not a programming language.

The programming languages JavaFX™ [38] and Cω [2] offer sequences, or streams, as array-like type constructors for variables with multiplicities greater than 1. Like arrays, sequences are reified multitudes of objects; however, unlike arrays, they are immutable and have value semantics. Sequences cannot be nested — any attempt to do so results in a flat sequence. null in the context of a sequence means the empty sequence and a scalar value means a singleton sequence, so that both can be assigned to a sequence-typed variable. In Cω, a stream can occur as the receiver of a member access; this access is then mapped over the elements of the stream, yielding a stream of the member type (so that chained member accesses on streams are possible). While this generalized member access has the same semantics as corresponding expressions in OCL and Alloy, the suitability of streams (which have been subsumed by iterators in C# 3.0 [4]) for implementing relationships to many objects is limited by their immutability.

The semantics of our static multiplicity *option* is somewhat similar to using the Option class in Scala [29]: a receiver of type Option can be None, in which case applying a function (using map or flatMap) produces None. Similarly, a function can be applied to a collection (again using map or flatMap), resulting in a collection of the same type, containing the return values. The main difference to object-oriented programming with multiplicities as put forward here is that we use no container types, but instead separate type from multiplicity, avoiding the awkward dominance of the container type over the content type [37] imposed by wrappers such as Option and collections. Another difference is that in object-oriented programming with multiplicities as we implemented it, the use of flatMap to apply functions to *option* and *any* multiplicities is implicit.

Ungar and Adams have recently presented a parallel programming language Ly that offers so-called *ensembles* as an alternative to collections [39]. Ensembles accommodate member objects that, when the ensemble is sent a message, all respond in parallel. However, unlike our multitudes of objects, an ensemble in Ly is a first-class object, and a singleton ensemble is different from the object that it contains. Since Ly is untyped, runtime checks are required to avoid that an ensemble contains itself (which may lead to infinite recursion when a message received by an ensemble is forwarded to itself). Also, empty ensembles are currently not integrated seamlessly, and demand further dynamic checks. It seems that the multiplicities described in this paper would solve at least some of the problems incurred by ensembles (but notably not those related to parallelism).

While implementing relationships to many objects using collections (or similar

reifications of multitudes) is by far the most commonly used pattern [26], automatic mappings from object-oriented models to programs may introduce other, more sophisticated patterns [18]. Both are however challenged by integrating relationships in object-oriented programming as a native concept.

As far back as 25 years ago, Rumbaugh argued for the lifting of the field-and-collection based relationship encodings of object-oriented programs to the level of a first class language construct [34]. For this purpose he introduced relations as instances of a special class Relation that has fields holding a relation declaration (i.e., the types of the participants, role names, cardinalities, etc.), as well as a field holding the extension of the relation (i.e., its tuples). Unlike in many other approaches that followed, an instance of Relation represents a relation, not a tuple; standard operations Rumbaugh defined on these instances included the adding and removal of tuples, indexed access to tuples of the relation, and scanning of the relation (iterating over its tuples). Later, Rumbaugh also added propagation attributes to relations which allowed the controlled recursive propagation of certain method invocations through object graphs [35]; however, this is not to be confused with our lifting of method invocations from single objects to multitudes of objects. While Rumbaugh's proposals amount to embedding a native implementation of (parts of) a relational database system in object-oriented programs, our approach of implementing to-many references is lightweight. Also, our relationships (represented by multitudes of objects) are not first-class.

Østerbye picked up Rumbaugh's proposals and presented a Smalltalk-based association compiler that can choose between internal and external implementations of relationships [32]. An internal implementation keeps the information which other objects an object is related to local to the object, whereas an external implementation uses first class relationship objects for this purpose. Independent of the implementation choice, Østerbye, like Rumbaugh before him, offers role-based and association-based access to relationships. However, in his role-based access protocol, he distinguishes between to-one and to-many relationships, continuing the discontinuity we want to rid programming of. This discontinuity is preserved in Østerbye's subsequent work [31], in which he leaves the untyped realm of Smalltalk to present a library-based approach for C#. In his library, association classes are complemented by role classes providing for internal implementation of relationships. However, given the fundamental meaning relationships have in most problem domains, we argue for a native, rather than a library-based, integration of relationships.

Bierman and Wren's RelJ is based on a formalized notion of relationships as first class types whose instances, called relationship instances, are tuples [3]. These tuples, which — like objects — can have state and behaviour, are created and returned by adding a pair of objects to a relationship. Navigation of a relationship always results in a set having value semantics, making the result of navigation covariant with the target type of the navigation [3]. However, sets cannot be the source of navigation, so that navigation cannot be chained as in our approach. Bierman and Wren also suggest how multiplicities can be restricted statically, using *one* (for $[0, 1]$, analogous to our *option*) and *many* (for $[0, *]$, analogous to our *any*) annotations; the invariant imposed by *one* is then enforced by changing the semantics of adding to a relationship with that of replacing an instance of a relationship (destructive update, or assignment). By contrast, we have restricted the additive update (+=) to *any* multiplicities, and require a downcast from *any* to *option* for an assignment to *option*, protecting us from a silent change of behaviour when a multiplicity is changed from *any* to *option*.

The relationship aspects of Pearce and Noble use the intertype declarations of AspectJ to shift the bookkeeping necessary for maintaining relationships between objects from the objects to relationships [33]. The relationships are coded as aspects which can carry additional, relationship-specific behaviour. Class definitions remain ignorant of the relationships for which they supply the participants, which is considered an increase in the separation of concerns. This separation goes too far, however, when an object needs access to others it is related to — in that case, it has to query the relationship it was to be kept unaware of.

In the language Rumer, references to objects are completely expelled from so-called entity types (conventional classes), and objects are related exclusively through relationship types [1]. It follows that, analogous to the relationship aspects of Pearce and Noble [33], only relationships know which entities are related (referred to as stratification in [1]). Entity and relationship types have associated extent types which are instantiated and populated explicitly by the programmer. Relationships can be nested, and relationship extents can be owned by relationships, so that they cannot escape the owning relationship. While owned relationship extents bear some resemblance to our multiplicities (which likewise cannot be aliased), the whole approach seems rather heavy weight — in particular, with all knowledge about relationships fully encapsulated in relationships (so that objects are ignorant of whether and how they are related), much of an application's logic (including that captured in most methods) has to be moved to relationships, with objects being degraded mostly to passive data containers with identity. This means a fundamental paradigm shift for object-oriented programming, and migrating an existing application to the concepts embodied in Rumer will amount to a major redesign effort.

8 Future Work

8.1 Integrating NonNull

As noted in Section 6.2.2, introduction of multiplicity *one* would help avoid an unpleasant restriction concerning the access of *bare* members via *option* receivers. The multiplicity *one* is equivalent to annotating a type use as being *NonNull* like in, for example, the Checker framework [11]. Additionally, (@one) can be used as a cast on an expression. Fähndrich and Leino showed how *NonNull* can be implemented to handle initialization correctly, introducing the notion of raw types [15]. This solution has been implemented for a previous version of JastAddJ [14]. A natural next step for us is thus to extend our implementation of multiplicities with this solution, supporting multiplicity *one*. We expect this to allow us to replace the multiplicity of most *bare* variables with *one*, and hence to reduce the number of casts substantially, as *one* expressions can safely be used as arguments to library methods requiring bares, and *bare* members (value types!) can safely be accessed on *one* receivers. Additionally, by adding type annotations, as introduced in Java 8, @NonNull annotations can be represented by *one* multiplicities, and be typechecked by the compiler.

8.2 Qualified Access

As noted in Section 4.1, a collection C used in an @any(C) annotation must have a single type parameter representing the type of the elements of the collection. This requirement excludes maps from a key type to a value type (such as HashMap<K, V>).

Not excluded, but not especially supported are indexed collections (like ArrayList<E>), which are special maps (with positive integers as keys): read access of the i^{th} object to which an expression e with multiplicity *any(List)* evaluates currently requires the clumsy workaround [[e]].get(i); write access is even clumsier (not shown here). For qualified access of the objects among a multitude, lists and maps can be generalized to associative arrays, effectively implementing the qualified associations of UML [28]. However, we have not yet investigated the language extensions this would require.

8.3 Case Studies on Modelling and Grammar Frameworks

Our current case study focuses on making use of multiplicities for ordinary Java code. Another interesting focus for case studies would be to focus on modelling frameworks such as EMF, where an API is generated from a metamodel expressing relations with cardinalities. It would be interesting to investigate how multiplicities could be used to simplify both the API and its usage. Furthermore, an interesting avenue of research would be to investigate to what extent metamodels can be automatically computed from code using multiplicities, reducing the gap between models and code.

In a similar manner, it would be interesting to investigate how multiplicities can simplify abstract syntax tree APIs, as generated by many compiler tools from EBNF-like formalisms. Here, there is a natural match between the typical *child*, *optional* and *list* constructs and the *one*, *option* and *any* multiplicities.

8.4 Refactoring to Multiplicities

In our experiment described in Section 6 we refactored JUnit manually from ordinary Java code to code using multiplicities. An interesting opportunity for further research is to design automated refactorings for this purpose. Based on the current case study we can see that most of these refactoring cases are fairly simple (they are related to a Change Declared Type refactoring), but also that there are a number of challenges that need to be addressed to find a general refactoring approach.

9 Conclusion

Letting expressions evaluate to any number of objects (rather than just one), and handling multitudes of objects that are not reified as one object, means a departure from object-oriented programming as we know it. In this paper, we have picked up a proposal for implementing object-oriented programming with multiplicities presented at last year's Onward! conference [37], and turned it into a fully functional compiler of Java 7 that can handle multitudes of objects as proposed. We tested this compiler by changing the source code of JUnit 4.0 so that it utilizes multiplicities, and by using the binaries produced by the compiler in place of the original binaries for running a number of different open source test suites on their programs under test. Functionally, both binaries are equivalent; furthermore, observed performance measures suggest that using multiplicities in JUnit 4.0 imposes only minor penalties. At the same time, a detailed analysis of the changes performed on the JUnit sources suggests that programs can indeed be simplified using multiplicities, avoiding many of the peculiarities imposed by using collections as containers of multitudes.

References

[1] Stephanie Balzer and Thomas R. Gross. "Verifying Multi-object Invariants with Relationships". In: *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*. Ed. by Mira Mezini. Springer, 2011, pp. 358–382. DOI: 10.1007/978-3-642-22655-7_17.

[2] Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. "The Essence of Data Access in Cω". In: *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*. Ed. by Andrew P. Black. Springer, 2005, pp. 287–311. DOI: 10.1007/11531142_13.

[3] Gavin M. Bierman and Alisdair Wren. "First-Class Relationships in an Object-Oriented Language". In: *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*. Ed. by Andrew P. Black. Springer, 2005, pp. 262–286. DOI: 10.1007/11531142_12.

[4] Gavin M. Bierman, Erik Meijer, and Mads Torgersen. "Lost in translation: formalizing proposed extensions to C#". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel et al. ACM, 2007, pp. 479–498. DOI: 10.1145/1297027.1297063.

[5] Stephen M. Blackburn et al. "Wake up and smell the coffee: evaluation methodology for the 21st century". In: *Commun. ACM* 51.8 (2008), pp. 83–89. DOI: 10.1145/1378704.1378723.

[6] Jordi Cabot and Martin Gogolla. "Object Constraint Language (OCL): A Definitive Guide". In: *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Ed. by Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. Springer, 2012, pp. 58–90. DOI: 10.1007/978-3-642-30982-3_3.

[8] Peter P. Chen. "The Entity-Relationship Model - Toward a Unified View of Data". In: *ACM Trans. Database Syst.* 1.1 (1976), pp. 9–36. DOI: 10.1145/320434.320440.

[9] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6 (1970), pp. 377–387. DOI: 10.1145/362384.362685.

[10] Steve Cook and John Daniels. *Designing Object Systems: Object-oriented Modelling with Syntropy*. Prentice Hall, 1994.

[11] Werner Dietl et al. "Building and using pluggable type-checkers". In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. Ed. by Richard N. Taylor, Harald Gall, and Nenad Medvidovic. ACM, 2011, pp. 681–690. DOI: 10.1145/1985793.1985889.

[12] Torbjörn Ekman and Görel Hedin. "The JastAdd system - modular extensible compiler construction". In: *Sci. Comput. Program.* 69.1-3 (2007), pp. 14–26. DOI: 10.1016/j.scico.2007.02.003.

[13] Torbjörn Ekman and Görel Hedin. "The JastAdd extensible Java compiler". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel et al. ACM, 2007, pp. 1–18. DOI: 10.1145/1297027.1297029.

[14] Torbjörn Ekman and Görel Hedin. "Pluggable checking and inferencing of non-null types for Java". In: *Journal of Object Technology* 6.9 (2007), pp. 455–475. DOI: 10.5381/jot.2007.6.9.a23.

[15] Manuel Fähndrich and K. Rustan M. Leino. "Declaring and checking non-null types in an object-oriented language". In: *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*. Ed. by Ron Crocker and Guy L. Steele Jr. ACM, 2003, pp. 302–312. DOI: 10.1145/949305.949332.

[16] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[17] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically rigorous java performance evaluation". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel et al. ACM, 2007, pp. 57–76. DOI: 10.1145/1297027.1297033.

[18] Dominik Gessenharter. "Implementing UML Associations in Java: A Slim Code Pattern for a Complex Modeling Concept". In: *Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages*. RAOOL '09. Genova, Italy: ACM, 2009, pp. 17–24. ISBN: 978-1-60558-549-9. DOI: 10.1145/1562100.1562104.

[19] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[20] Görel Hedin. "Reference Attributed Grammars". In: *Informatica (Slovenia)* 24.3 (2000).

[21] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2011. ISBN: 978-0-262-01715-2.

[22] Barbara Liskov and Jeannette M. Wing. "A Behavioral Notion of Subtyping". In: *ACM Trans. Program. Lang. Syst.* 16.6 (1994), pp. 1811–1841. DOI: 10.1145/197320.197383.

[23] Microsoft Corporation. *C# Language Specification v1.2*. http://download.microsoft.com.

[24] Maurice Naftalin and Philip Wadler. *Java generics and collections*. O'Reilly, 2006. ISBN: 978-0-596-52775-4.

[25] James Noble, Jan Vitek, and John Potter. "Flexible Alias Protection". In: *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*. Ed. by Eric Jul. Springer, 1998, pp. 158–185. DOI: 10.1007/BFb0054091.

[26] James Noble. "Basic relationship patterns". In: *Pattern Languages of Program Design 4*. Addison-Wesley, 2000, pp. 73–89.

[27] Object Management Group. *Object Constraint Language Version 2.2*. http://www.omg.org/spec/OCL/2.2.

[28] Object Management Group. *UML Superstructure V2.2*. http://www.omg.org/spec/UML/2.2/Superstructure.

[29] Martin Odersky. *The Scala Language Specification*. 2009.

[30] Jesper Öqvist and Görel Hedin. "Extending the JastAdd extensible Java compiler to Java 7". In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013*. Ed. by Martin Plümicke and Walter Binder. ACM, 2013, pp. 147–152. DOI: 10.1145/2500828.2500843.

[31] Kasper Østerbye. "Design of a Class Library for Association Relationships". In: *Proceedings of the 2007 Symposium on Library-Centric Software Design*. LCSD '07. Montreal, Canada: ACM, 2007, pp. 67–75. ISBN: 978-1-60558-086-9. DOI: 10.1145/1512762.1512769.

[32] Kasper Østerbye. "Associations as a Language Construct". In: *TOOLS Europe 1999: 29th International Conference on Technology of Object-Oriented Languages and Systems, 7-10 June 1999, Nancy, France*. IEEE Computer Society, 1999, pp. 224–235. DOI: 10.1109/TOOLS.1999.779015.

[33] David J. Pearce and James Noble. "Relationship aspects". In: *Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD 2006, Bonn, Germany, March 20-24, 2006*. Ed. by Robert E. Filman. ACM, 2006, pp. 75–86. DOI: 10.1145/1119655.1119668.

[34] James E. Rumbaugh. "Relations as Semantic Constructs in an Object-Oriented Language". In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), Orlando, Florida, USA, October 4-8, 1987, Proceedings*. Ed. by Norman K. Meyrowitz. ACM, 1987, pp. 466–481. DOI: 10.1145/38765.38850.

[35] James E. Rumbaugh. "Controlling Propagation of Operations Using Attributes on Relations". In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'88), San Diego, California, USA, September 25-30, 1988, Proceedings*. Ed. by Norman K. Meyrowitz. ACM, 1988, pp. 285–296. DOI: 10.1145/62083.62109.

[36] James E. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1990.

[37] Friedrich Steimann. "Content over container: object-oriented programming with multiplicities". In: *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*. Ed. by Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld. ACM, 2013, pp. 173–186. DOI: 10.1145/2509578.2509582.

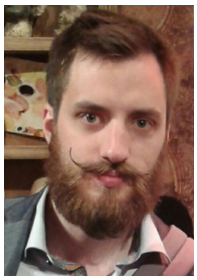
[38] K. Topley. *JavaFX Developer's Guide*. Developer's Library. Pearson Education, 2010. ISBN: 9780321648952.

[39] David Ungar and Sam S. Adams. "Harnessing emergence for manycore programming: early experience integrating ensembles, adverbs, and object-based inheritance". In: *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, 2010, pp. 19–26. DOI: 10.1145/1869542.1869546.

[40] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. "Higher-Order Attribute Grammars". In: *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*. Ed. by Richard L. Wexelblat. ACM, 1989, pp. 131–145. DOI: 10.1145/73141.74830.

About the authors

Friedrich Steimann is head of Programming Systems at the Fernuniversität in Hagen, Germany. He conducts research on languages and tools supporting programmers' productivity. Contact Friedrich at steimann@acm.org.



Jesper Öqvist is a PhD student in Computer Science at Lund University, Sweden. He is interested in construction and application of modular compilers. He is the current maintainer of the JastAddJ compiler. Contact Jesper at jesper.oqvist@cs.lth.se.

Görel Hedin is professor in Computer Science at Lund University, Sweden. Her research interests include object-oriented languages and design, extensible language implementation, and agile methodology. Contact Görel at gorel.hedin@cs.lth.se.

Acknowledgments This work was in part financed by the Swedish Research Council under grant 621-2012-4727.