JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 6, No. 3, March–April 2007

Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance

Christian Heinlein, Dept. of Computer Science, University of Ulm, Germany

Open types are presented as a simple yet powerful data model for statically typed procedural and object-oriented programming languages, that overcomes the limitations of the traditional record-oriented model. The basic idea is to separate type definitions from the definitions of their attributes in order to allow incremental definitions of the latter. Furthermore, bidirectional relationships are introduced as pairs of mutually inverse attributes whose values will be kept consistent automatically. Finally, anonymous and automatic attributes are presented as a simple extension of attributes that unifies the concepts of aggregation, inheritance (including subtype polymorphism), and user-defined type conversions. Since open types do not possess methods in the object-oriented sense, global virtual functions are used as an alternative and more flexible means to define behaviour, while modules are employed to achieve information hiding and type-safe separate compilation. The basic implementation ideas of open types as a language extension for C++ are described.

1 INTRODUCTION

Since the advent of procedural programming languages in the 1960s, data structures appearing in programs are typically modeled as *records*, and even the object-oriented notion of a *class* is basically a record equipped with accompanying procedures or methods. Even though data modeling based on records, e. g., modeling a person as a record possessing *fields* (or *members*) such as name, date of birth, address (which might itself be a record), etc., is rather natural and straightforward, a severe limitation of records is the fact that they are *fixed*: Once a record type has been defined, the set of its fields is invariably determined, and all its instances will possess exactly these fields.

Variant records in procedural languages and *subclasses* in object-oriented languages provide some more flexibility in this regard, but a particular type or class still remains fixed once it has been defined. So, even though it is possible, for instance, to "extend" a given class `Person` by defining a subclass `PersonWithSsno` in order to model persons with a social security number (ssno), the base class `Person` actually remains unchanged (so the term "extend" is quite misleading). This is particularly problematic when an existing program (e. g., a person management system) shall be extended later in an unanticipated way: If the original source code is not available or shall not be modified for reasons of modularity, introducing a subclass

Cite this article as follows: Christian Heinlein: *"Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance"*, in Journal of Object Technology, vol. 6, no. 3, March – April 2007, pp. 101–151, http://www.jot.fm/issues/issue_2007_03/article3

of `Person` does not help since the original code still creates instances of the original class.

To overcome these kinds of problems, aspect-oriented programming languages provide so-called *inter-type member declarations* [22] or *introductions* [32] in order to actually *extend* an existing class definition with new data fields (and methods) in a *modular* way (i. e., without needing to touch existing source code), *without* defining a new class for that purpose. Interestingly, the possibility to truly extend the set of data fields of a class later actually makes the original possibility to define a class with data fields superfluous: It is always possible in principle to define a class empty, i. e., without any fields, and to add them incrementally later.

If a particular class has been extended with additional data fields multiple times, typically by different "aspects" of a program, it might happen that many instances of the class do not actually need *all* the fields introduced that way. For example, the social security number might be stored only for persons having an employment, while information about a person's spouse and children might be needed only for those receiving child benefit. But even for "conventionally" defined data structures, where all data fields are defined at once, it might happen that fields of particular instances remain unused simply because particular information is either unknown or irrelevant. Depending on the total number of unused data fields in a program, it might be worthwhile to have an underlying data representation format that stores only those data fields of an object which are actually used, similar to the way database systems optimize storage of objects containing null values.

Finally, *bidirectional relationships* between data types, e. g., between a person and the cars it owns, are hard to express naturally with record-based data models, since conceptually they do not belong to either type, but rather constitute entities in their own right. In order to implement them in procedural or object-oriented programming languages, it is necessary to model them as dual pairs of data fields of the participating record types, one for each "direction" of the relationship, which must be kept consistent explicitly. If, for instance, a car changes its owner, not only must its `owner` field be changed accordingly, but also the `cars` fields of both the old and the new owner must be modified to correctly reflect the new situation.

To overcome all these limitations of traditional record and class types mentioned above, *open types* will be presented in this paper as an alternative data model for statically typed procedural and object-oriented programming languages. Beforehand, however, Sec. 2 gives some introductory remarks about the decision to provide open types as a *syntactical extension* of an existing language (viz. C++) instead of developing either a completely new language or a library for an existing one. Afterwards, Sec. 3 briefly presents *null values* as a generally useful concept which is particularly relevant for the design of open types.

After these preparations, Sec. 4 introduces the basic notion of open types, together with single- and multi-valued *attributes*, while Sec. 5 describes bidirectional *relationships* as pairs of mutually inverse attributes whose values will be kept con-

sistent automatically. Section 6 presents *anonymous* and *automatic attributes* as a simple extension of attributes that unifies the concepts of aggregation, inheritance (including subtype polymorphism), and user-defined type conversions.

Since open types are only one of two fundamental concepts of so-called *advanced procedural programming languages*[1], the complementary concept of *global virtual functions* published earlier [14, 16] is recapitulated in Sec. 7, followed by a description of *modules* to support encapsulation, information hiding, and type-safe separate compilation in Sec. 8. To give a coherent example for the application of these concepts in concert, Sec. 9 presents a solution of the well-known *expression problem* [35].

The basic ideas to implement open types, both on the linguistic level and in terms of low-level memory management and garbage collection, are presented in Sec. 10, before the paper is concluded with a discussion of related work in Sec. 11 and a general conclusion in Sec. 12.

2 DESIGN ALTERNATIVES

Generally speaking, there are at least three different possibilities for providing a new programming language concept: It might be implemented either as a *library* for an existing language, possibly accompanied by a framework of design patterns or coding rules (e. g., JBoss AOP [17], which implements aspect-oriented concepts as a Java library), or as a *syntactical extension* of an extension language (e. g., AspectJ [22], which extends Java with aspect-oriented language constructs), or in a completely *new language*.

After careful evaluation of the pros and cons of these alternatives, it has been decided to implement open types as a precompiler-based language extension for the following reasons:

- Providing them as a pure library framework for a statically typed procedural or object-oriented language turned out to be too complex and unwieldy to use. In fact, the code generated by the precompiler for open types and attributes is so verbose and complicated that no programmer would volunteer to write it manually.

- The effort for developing a completely new *advanced procedural programming language* based on open types, global virtual functions, and modules, has been regarded too high for now, even though it is still considered a long-term goal.

- Even though a precompiler-based implementation of a language extension has some well-known limitations, e. g., with respect to proper diagnosis of and recovery from syntax errors, it allows to construct a working environment

[1] See www.informatik.uni-ulm.de/rs/mitarbeiter/ch/apples.

for the new language concept, which can be used to gain valuable practical experience with it, in a comparatively short amount of time.

The decision to actually use C++ as the base language has been influenced, amongst others, by the following considerations:

- Since open types are designed as a statically type-safe concept, the base language must at least support the notion of static types and provide a sufficient amount of static type checking. Therefore, purely dynamically typed languages such as Smalltalk or Lisp have been ruled out. The fact that static type checking in C++ can always be circumvented by dirty tricks (such as taking the address of an object, casting it to a different pointer type, and dereferencing it) has been accepted as a minor drawback, since *accidental* violations of the static type system are detected by the compiler.

- In contrast to (more) pure object-oriented languages such as Eiffel or Java, C++ is actually a multi-paradigm language [24] that supports traditional procedural programming as well. Since open types and global virtual functions are more related to traditional record types and procedures than to classes and methods, they fit in more naturally with a language providing these concepts than with a bare object-oriented language. Since the resulting language called C+++ supports object-oriented programming alongside with traditional and advanced procedural programming, a programmer can (and should) select a particular subset of the language which fits his needs and personal preferences. In particular, when using open types, all stuff related to classes with all its accompanying complexity becomes dispensable.

- C++ provides a rich set of independently useful concepts (one might also say that it is conceptually overloaded), such as templates, function and operator overloading, default arguments, etc., which are not only useful for application programmers, but also for the code generated by the precompiler, which, for instance, makes heavy use of templates.

Despite these arguments for C++ (and against pure object-oriented languages), it should be noted that the basic concept of open types is actually language-independent and might in fact be incorporated into any statically typed procedural or object-oriented programming language.

3 NULL VALUES

The concept of *null values*, i. e., special "values" representing the absence of any real value, is provided in various forms by some programming languages, e. g., `nil` in Lisp, `None` in Python, `null` in Java, or NaN (not a number) in languages providing IEEE floating point arithmetics [12]. While the latter can be used with well-defined

semantics just like real values (where the value of an expression containing one or more NaN operands is NaN, too), using any of the former in an operation usually leads to a run time error. Furthermore, statically typed languages typically do not provide null values for *all* types of the language, but mainly for reference types.[2]

As has been shown in more detail in [15], however, providing null values with well-defined behaviour as a general concept for all types of a language yields several benefits and naturally solves some otherwise intractable problems, in particular:

- If a variable is not explicitly initialized, it naturally possesses *no value*, i. e., null, instead of an undefined value or a default value such as zero.

- If a function does not explicitly return a value, because it does not contain or execute a `return` statement, it naturally returns *no value*, i. e., null.

In fact, many dynamically typed languages actually exhibit such a behaviour. On the other hand, attempts to statically detect these situations at compile time (as, e. g., in Java) are necessarily limited to conservative flow analyses due to the theoretical undecidability of the underlying problems, which frequently lead to annoying "false" error messages.

To incorporate such a general notion of null values into C++, wrapper types called `integer`, `character`, etc. are provided for the built-in types `int`, `char`, etc. whose parameterless constructor yields a null value which is different from any real value of the type (including zero). This implies that variables of such a type which are not explicitly initialized, are implicitly initialized to null. To conveniently check whether a given value is null, an implicit conversion to a Boolean value is provided that returns `true` for all real values (including zero!) and `false` for a null value. To achieve that functions that do not explicitly return a value implicitly return null, a corresponding return statement is added by the precompiler at the end of every function body.

For the design of open types, only the fact that the parameterless constructor of a type logically returns nothing, will be essential, while the other aspects of null values might be considered independently useful.

4 OPEN TYPES AND ATTRIBUTES

Based on the preparations of the previous section, this section introduces the core concepts of *open types* and *attributes*, using the simple conceptual data model depicted in Fig. 1 as an example, while subsequent sections about *bidirectional relationships* and *anonymous and automatic attributes and relationships* will describe some important extensions of this basic model.

[2]Besides IEEE NaN values for floating point types, C#'s `Nullable` types constitute a partial exception; however, instead of actually providing null values for all types of the language, there is a separate nullable type `T?` for each non-nullable type `T`.

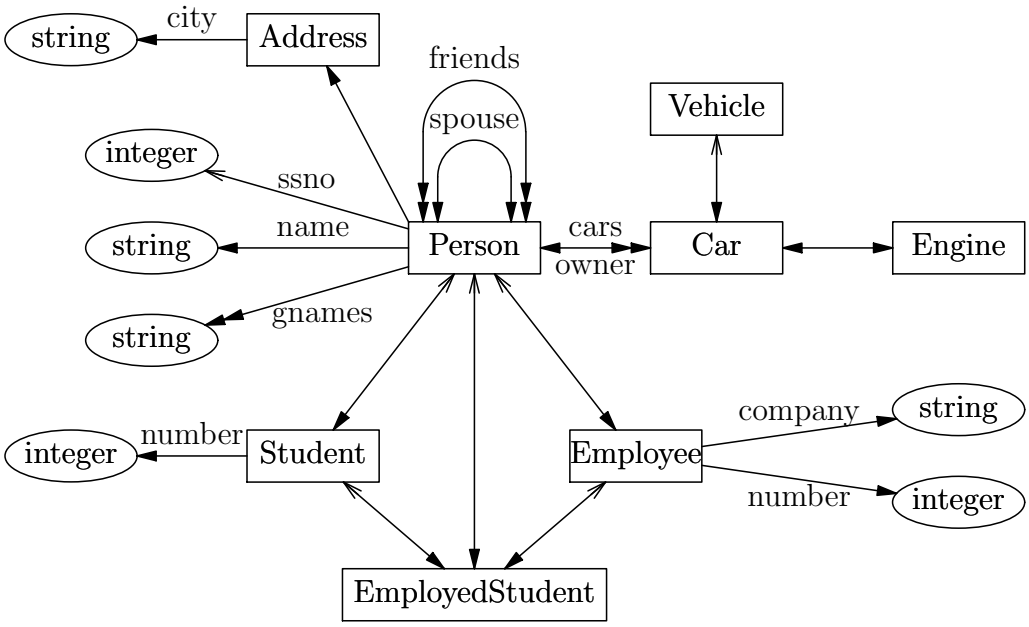


Figure 1: Conceptual data model

Type and Attribute Definitions

An *open type* is simply defined by declaring its name with the keyword `typename`, e. g.:

```
typename Person;
typename Car;
```

This corresponds to drawing the rectangles labeled `Person` and `Car` in the graphical data model.

Afterwards, a *single-valued attribute* such as `name`, corresponding to a data field in record notion, can be defined by declaring it as a kind of mapping from `Person`s to `string`s (where the latter type constitutes a predefined *atomic* type depicted by an ellipse instead of a rectangle in order to distinguish it from user-defined open types):

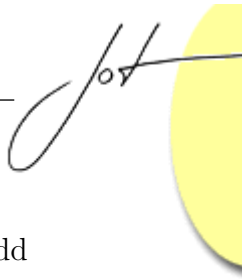
```
Person -> string name;
```

This corresponds to drawing an arrow labeled `name` from `Person` to `string` in the graphical data model. Simultaneously, the right hand side of the definition (`string name;`) looks identical to a C++ (member) variable definition.

Similarly, a *multi-valued attribute* such as `gnames` (given names), corresponding to a data field whose type is an array or container type, is defined by using a double instead of a single arrow to indicate the multi-valuedness:

```
Person ->> string gnames;
```

Again, this corresponds directly to drawing a corresponding arrow in the graphical



data model.

Since type and attribute definitions are *separated*, it is easily possible to add new attributes of a type on demand, either in the same or in a different translation unit, without needing to change the original type definition. It is even possible to dynamically load modules containing additional attribute definitions for a type, even after objects of this type have been created.

Objects and References

The value of an open type variable or expression is either an internal pointer or *reference*[3] to an *object* of that type or a *null reference*, in the same way as the value of a variable or expression whose type is a Java class is either a reference to an object of that class or the special value `null`. As a consequence, copying an open type value simply means to copy the reference instead of the referenced object, while comparing two values simply means to compare their references.

In contrast to Java, however, where `null` constitutes a generic value compatible with all reference types, there is a separate null value for each open type that can be obtained by calling its parameterless constructor. To check whether a given value is null, an implicit conversion to a Boolean value is provided that returns `true` if the value actually references an object and `false` if it constitutes a null reference (cf. Sec. 3).

Constructors and Mutators

To create, initialize, and modify objects of an open type `T`, the following *constructors* and *mutators* are provided.

As already mentioned, the *parameterless constructor* `T()`, which might either be called explicitly or is called implicitly for variables of type `T` which are not initialized explicitly [33], returns the *null reference* of type `T`, i. e., a reference to *no object*. In contrast, the *attribute-initialization constructor* `T(@attr, val)` creates a distinct *new object* of type `T`, i. e., an object that is different from null and any other object, initializes its attribute `attr` with value `val`, and returns a reference to it. Since open types are designed to be statically type-safe, `attr` must have been declared as an attribute of type `T`, and `val` must be assignment-compatible with its target type.

Similarly, the *attribute mutator* `obj(@attr, val)` modifies the object (referenced by) `obj` by setting the value of its attribute `attr` to `val` (if `attr` is a single-valued attribute) or by appending `val` to its values of attribute `attr` (if `attr` is a multi-valued attribute). Expressed differently, `obj(@attr, val)` always adds `val` to `obj`'s values of `attr`, after discarding any previous value if `attr` is single-valued.[4] Again,

[3] The term "reference" is used in its general meaning here, which is somewhat different from the C++ notion of references [33].

[4] For multi-valued attributes, there are also variants of the mutator to remove a previously added value and to insert a new value at a particular position.

to ensure static type safety, `attr` must have been declared as an attribute of `obj`'s (static) type, and `val` must be assignment-compatible with its target type. As a result of the operation, the object (reference) `obj` is returned, which allows straightforward combinations of a constructor call with one or more mutator calls to create an object with multiple initial attribute values, e. g.:

```
Person p = Person(@name, "Hoare")
 (@gnames, "Charles")(@gnames, "Anthony")(@gnames, "Richard");
```

Here, the constructor call `Person(@name, "Hoare")` creates a new `Person` object, initializes its attribute `name` with the string `"Hoare"`, and returns (a reference to) the object. This object (reference) is directly used in the mutator call `...(@gnames, "Charles")` which initializes its attribute `gnames` with `"Charles"` and returns the same object (reference). This is again used in and returned by the subsequent mutator calls `...(@gnames, "Anthony")` and `...(@gnames, "Richard")` which in turn add the strings `"Anthony"` and `"Richard"` to the values of attribute `gnames`. Finally, the object (reference) returned by the last mutator call is assigned to the `Person` variable `p`.

Of course, it is also possible to perform constructor and mutator calls separately, e. g.:

```
Person p = Person(@name, "Hoare");
p(@gnames, "Charles");
p(@gnames, "Anthony")(@gnames, "Richard");
```

To create an *empty object*, i. e., a distinct object which is different from null and any other object, but does not possess any attribute values yet, the *empty constructor* `T(@)` can be used, which is called with a pseudo-argument `@` to distinguish it from the parameterless constructor `T()`. Therefore, a call `T(@attr, val)` to the attribute-initialization constructor is actually just a shorthand for `T(@)(@attr, val)`, i. e., a call to the empty constructor followed by an appropriate mutator call.

In addition to these predefined constructors of open types, it is possible to define arbitrary *user-defined constructors*, e. g.:

```
// Create person with given name g and name n.
Person (string g, string n) {
  return Person(@name, n)(@gnames, g);
}
```

In contrast to normal C++ constructors (and constructors in other object-oriented programming languages), which must be defined (or at least declared) inside their class and must not explicitly return anything, but rather initialize the implicitly available object `this`, user-defined constructors of open types are much like ordinary (global) functions whose result type and name coincide (and therefore only one of them is specified in their definition). In particular, there is no implicitly available object `this`, and the constructor must explicitly (create and) return an object, typically by calling one of the predefined constructors. Furthermore, just like attributes,

constructors can be defined *incrementally* on demand.

Attribute Inspections

To inspect the attribute values of a given object, the *attribute inspection operator* `@` can be used, quite similar to the way the dot operator is used to access class members in C++ and other languages, e. g., `p@name` or `p@gnames`.[5]

For a single-valued attribute such as `name`, its current value is returned, i. e., the value that has been set for this attribute by the most recent mutator or attribute-initialization constructor call for this object. If none of these operations has been performed for the object yet, i. e., the attribute does not possess any value, *nothing* is returned conceptually, i. e., a null value of the attribute's target type (i. e., `string` in the current example) that is obtained by calling its parameterless constructor (cf. Sec. 3).[6]

If a multi-valued attribute such as `gnames` is inspected with the `@` operator, the values added to this attribute by all mutator (and attribute-initialization constructor) calls performed for this object so far are returned as a (possibly empty) *ordered sequence*. Even though it is possible to grasp such a sequence as a whole, it is typically processed element by element using a tailored iteration statement, e. g.:

```
for (string g : p@gnames) cout << g << " ";
```

This prints `p`'s given names in the order in which they have been added, i. e., `Charles Anthony Richard`. Alternatively, it is possible to directly inspect individual values of such a sequence by applying the standard index operator, e. g., `p@gnames[2]`, to obtain the second given name of `p`, i. e., `"Anthony"`. Similarly to inspecting the value of a non-existent single-valued attribute, inspecting a non-existent value of a multi-valued attribute by using an out-of-range index yields nothing, i. e., a null value that is obtained in the same way as described above. Therefore, expressions such as `p@gnames[0]` or `p@gnames[4]` will return a null string in the current example.

Similar to an attribute mutator operation, an attribute inspection `obj@attr` is type-correct, if `attr` has been declared as an attribute of `obj`'s static type, and the

[5] It would have been possible to reuse the existing dot or `->` operator for that purpose, too, just like existing keywords such as `typename` have been reused to avoid the introduction of new ones. Introducing a new operator, however, cannot create any incompatibilities with existing C++ code, while introducing a new keyword (e. g., `type`) would cause programs using this as an identifier to become invalid.

[6] If the target type of an attribute is a user-defined or built-in C++ type (such as `int`) that does not possess a distinct null value, its parameterless constructor would return a *default value* such as zero, which is conceptually quite different from *no value*. Therefore, such types should ideally be prohibited as target types of attributes, which could actually be enforced by the precompiler. For practical reasons, however, in particular to simplify the integration of C++ legacy code into C+++ programs, any kind of target type is allowed. By means of a compile-time switch, it is possible to specify whether the default value returned by the parameterless constructor shall be returned or an exception shall be raised when a non-existent attribute value of such a target type is accessed. If a type does not possess a public parameterless constructor, the former alternative will lead to a compile-time error.

result type of the operation is equal to the attribute's target type (for single-valued attributes) or a sequence of it (for multi-valued attributes), where a sequence is defined as a type-safe container similar to an array. In either case, the attribute inspection operator returns an *R-value* [33], i. e., a value which must not occur on the left hand side of an assignment operator. Therefore, attribute update operations must only be performed by mutator calls, not directly by assignments such as:

```
p@name = "Hoare"; // Syntax error!
```

If such an assignment would be allowed, the attribute inspection operator could not simply return a null value for a non-existent attribute, but would be forced to implicitly create the attribute in that case, i. e., to allocate space for it, in order to be able to return an *L-value* referring to it. This would lead to many unnecessary attribute creations in practice.

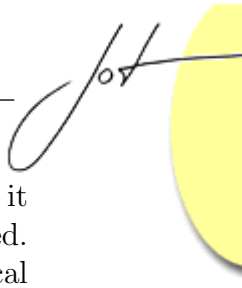
Inspecting and Modifying Null Objects

Trying to inspect or modify the "object" referenced by a null pointer is illegal in C++ and many other languages and usually leads to a run time error such as a `SIGSEGV` (segmentation violation signal) or a `NullPointerException`. In contrast, inspecting and modifying attributes of open types is well-defined even for null objects/references: While inspecting such an attribute is equivalent to inspecting a non-existent attribute, i. e., returns null, modifying such an attribute simply has no effect. The main reason for these unusual definitions is convenience, since they allow to omit many otherwise necessary checks. To test, for example, whether `p`'s name is `"Hoare"`, one can simply write `if (p@name == "Hoare")` – instead of `if (p && p@name == "Hoare")` – even if `p` might be null; if it actually is, `p@name` is null, too, and therefore, as expected, different from `"Hoare"`. Similarly, testing whether the third given name of `p` is `"Richard"`, one simply writes `if (p@gnames[3] == "Richard")`, without needing to check explicitly whether `p` is not null, whether it has given names at all, and whether it has a third given name. Simultaneously, programs tend to become more robust since inadvertently omitted checks will not lead to run time errors, but usually merely to unsatisfied conditions.

Similarly, the fact that mutator calls on null objects are silently ignored, frequently reduces the need to explicitly distinguish between real and null objects, and again, inadvertently omitting such distinctions does not lead to run time errors. Reference [15] contains a more detailed discussion of the pros and cons of this unusual definition.

Garbage Collection and Object Deletion

In contrast to normal C++ objects, which must be explicitly deleted by the programmer to reclaim their storage, objects of open types are automatically *garbage-collected* when they have become unreachable, quite similar to objects of classes in Java, Eiffel, Smalltalk, and many other programming languages.



In addition to and independently from this automatic storage reclamation, it is also possible to explicitly *delete* objects – even while they are still referenced. Although this might appear strange at first sight, there are reasonable practical examples where this is useful. To give an extreme one, if a person has died, it simply does not exist anymore, no matter whether there are still "references" to it. Similarly, if a car has been scrapped, it does not exist anymore, even though its (previous) owner might still reference it. In Sec. 6.7, another important application of explicit object deletion will be presented.

In contrast to C++, however, where the deletion of an object might lead to dangerous dangling pointers, deletion of an open type object causes all remaining references to the object to become null immediately and automatically. By that means, it is always possible to reliably detect that an object has been deleted. Furthermore, since null objects can be safely inspected and modified, too, neither run time errors nor undefined behaviour will occur if deleted objects are used without care. Since object deletions might be performed unexpectedly, this is another strong argument for the unusual definitions given in Sec. 4.5.

Enumerations and Variant Records

Open types can be used in a straightforward way to define *extensible* enumeration types by defining the enumeration values as constants initialized with (unique) empty objects, e. g.:[7]

```
// Initial definition.
typename Color;
const Color red(@), green(@), blue(@);
......

// Later extension.
const Color cyan(@), magenta(@), yellow(@);
```

Furthermore, the fact that attributes of an open type are actually *optional*, i. e., individual objects need not possess values for all attributes, can be exploited to model variant records without requiring additional language constructs, e. g.:

```
typename Expr;          // Arithmetic expression.
Expr -> integer val;    // Value of a constant expression.
Expr -> character op;   // Operator of a compound expression.
Expr -> Expr body;      // Body of a unary expression.
Expr -> Expr left;      // Left and right operand
Expr -> Expr right;     // of a binary expression.
```

When used properly, objects of type `Expr` either contain a value `val` or an operator `op` plus either a body `body` or two operands `left` and `right`. This is reflected by the following user-defined constructors:

[7] A C++ declaration such as `Color red(...);` is a shorthand notation for `Color red = Color(...);`

```
// Create constant expression with value v.
Expr (integer v) { return Expr(@val, v); }

// Create unary expression with operator o and body b.
Expr (character o, Expr b) { return Expr(@op, o)(@body, b); }

// Create binary expr. with operator o and operands l and r.
Expr (Expr l, character o, Expr r) {
  return Expr(@left, l)(@op, o)(@right, r);
}
```

Interestingly, `Expr` objects do not need a separate "tag field" encoding the particular kind of expression: If an object `x` possesses a value `val`, i. e., if `x@val` is not null, it is considered a constant expression; otherwise, if it possesses a body, i. e., if `x@body` is not null, it is considered a unary expression and `x@op` is expected to contain its operator; finally, if an object `x` possesses operands `x@left` and/or `x@right`, it is treated as a binary expression with operator `x@op`. It should be noted, however, that it is also possible to model data structures in a more object-oriented way, resembling type hierarchies with sub- and supertypes, as described in Sec. 6.3.

Since the underlying implementation will not consume space for unused attributes of an object (cf. Sec.10.1), using attributes "generously" is not a problem. In particular, it is not necessary to apply tricks such as reusing, e. g., `left` or `right` to store the body of a unary expression or to encode the operator of a compound expression in the attribute `val` in order to save space.

5 BIDIRECTIONAL RELATIONSHIPS

Basically, a *bidirectional relationship* between two open types is also a kind of mapping from one type to the other, with the additional possibility to directly access the *inverse mapping*. Since both of these mappings might be either single- or multiple-valued, there are four different kinds of relationships altogether, one to one, one to many, many to one, and many to many, expressed by corresponding bidirectional arrow symbols `<->`, `<->>`, `<<->`, and `<<->>`, respectively. Furthermore, there are two special kinds, i. e., *symmetric* one-to-one and many-to-many relationships, where both mappings coincide.

Relationship Definitions

To continue the example depicted in Fig. 1, the one-to-many relationship between `Person` and `Car` called `cars` resp. `owner` can be defined as follows:

```
Person owner <->> Car cars;
```

Reading this from left to right (and omitting the attribute name on the left side) yields a multi-valued attribute `cars` of type `Person`, while reading from right to left (and omitting the attribute name on the right side) yields a single-valued attribute

owner of type Car representing the inverse mapping:

```
Person ->> Car cars;
Car -> Person owner;
```

However, only by combining both attribute definitions into a single relationship definition as shown initially, they are actually treated as mutually inverse mappings, which means that a call to one of the mutators automatically implies a corresponding call to the other mutator with reversed roles.

For example, a mutator call such as p(@cars, c), adding c to p's sequence of cars, implies the call c(@owner, p), assigning p as c's owner, and vice versa. Furthermore, if c already possesses another owner q when either such call is made, c is first removed from the sequence of q's cars.

According to the rules for unidirectional attributes (cf. Sec. 4.5), the mutator calls p(@cars, c) and c(@owner, p) exhibit the following behaviour if p and/or c is null:

- If p is null, but c is not, p(@cars, c) has no effect per se, while c(@owner, p) sets the owner of c to null, i. e., removes the current owner q of c, if any, which implies the removal of c from q's sequence of cars. However, since either mutator call implies a call to the other, even p(@cars, c) will cause c to become owner-less.
- If c is null, but p is not, c(@owner, p) has no effect per se, while p(@cars, c) adds a null car to p's sequence of cars. Again, since either mutator call implies a call to the other, even c(@owner, p) will cause p to receive a null car.
- If both p and c are null, both p(@cars, c) and c(@owner, p) have no effect.

Similar rules exist for the other kinds of relationships, where a many-to-one relationship is equivalent to a one-to-many relationship with reversed roles. Even though the precise specification of these rules is somewhat lengthy in order to cover all possible combinations of null and non-null values, their intuitive meaning is simply expressed by the fact that the two mappings defined by a relationship are inverse to each other. To preserve this property, it is sometimes necessary to modify additional objects which are not directly mentioned in an update operation. For example, if a one-to-one relationship between two objects is established, up to two other objects might be involved in the operation which are currently associated with the objects in question.

Symmetric Relationships

Symmetric relationships such as spouse (one to one) and friends (many to many) are almost completely equivalent to ordinary relationships whose types and names are the same for both directions. Therefore, the right hand side of the definition is simply omitted, e. g.:

```
Person spouse <->;
Person friends <<->>;
```

According to the normal rules for relationships, a mutator call such as `p(@spouse, q)` or `p(@friends, q)`, declaring `q` as `p`'s spouse resp. friend, implies the inverse call `q(@spouse, p)` resp. `q(@friends, p)`, declaring `p` as `q`'s spouse resp. friend, and vice versa. The only exception to this rule is that the inverse call is suppressed if `p` and `q` refer to the same object, i. e., in that case `p` is declared only once as its own spouse (which is semantically strange, of course) resp. friend.

6 ANONYMOUS AND AUTOMATIC ATTRIBUTES AND RELATIONSHIPS

Anonymous Attributes and Relationships

Sometimes, the most appropriate name for an attribute is simply the name of its target type. Given, for example, an open type `Address`, an attribute of `Person` representing a person's address might simply be called `Address`, too, instead of inventing an artificial different name such as `address`. Viewed differently, such an attribute does not possess an explicit name (cf. the unnamed arrow between `Person` and `Address` in Fig. 1), but rather implicitly receives the name of its target type. Therefore, such attributes are called *anonymous*, and their declaration does not introduce a new name, e. g.:

```
typename Address;
Person -> Address;
```

In order to inspect and modify the values of an anonymous attribute, the name of its target type is used, e. g.:

```
// Address attributes.
Address -> string city;
......

// Initialize p's address.
Address a = Address(@city, "New York")(......);
p(@Address, a);

// Print p's address.
cout << p@Address@city << endl;
......
```

Similarly, if one or both names of a relationship definition are omitted, the corresponding type names can be used instead, e. g.:

```
typename Engine;
Car <-> Engine;
Car c = Car(@Engine, Engine(......));
```

Automatic Attributes and Relationships

If the arrow in an attribute declaration is followed by an exclamation mark, the attribute might be applied *automatically* on demand to perform an *implicit type conversion* from its source type (left of the arrow) to its target type (right of the arrow), e. g.:

```
Person ->! integer ssno;
```

This declares an `integer` attribute `ssno` for type `Person`, which can be used just like any other attribute, with the additional property that an expression of type `Person` is implicitly convertible to an `integer` value by automatically applying this attribute.

Similarly, it is possible to declare automatic relationships by adding an exclamation mark before or after the bidirectional arrow, depending on which direction of the relationship should be automatically applicable.

Modeling Type Hierarchies

Automatic one-to-one relationships can be exploited to model object-oriented type hierarchies with subtype polymorphism, without requiring any additional mechanisms. For example, a new type `Student` might be defined as a "subtype" of `Person` by declaring a one-to-one relationship between these types that is automatically applicable from the derived type to the base type:

```
// Declare Student as a "subtype" of Person.
typename Student;
Student <->! Person;
```

Typically, but not necessarily, such "subtype" relationships are anonymous. In the graphical data model (Fig. 1), the automatic direction of a relationship is indicated by an open instead of a filled arrow head.

If another regular attribute `number` denoting a student's matriculation number as well as a user-defined constructor accepting a person "part" and a matriculation number is defined, a student named Peter Clark with matriculation number 777 can be created and used as follows (employing the user-defined `Person` constructor introduced in Sec. 4.3):

```
// Matriculation number.
Student -> integer number;

// Create student with person part p and matriculation number m.
Student (Person p, integer m) {
  return Student(@Person, p)(@number, m);
}

// Create student.
Student s = Student(Person("Peter", "Clark"), 777);
```

```
// Print name and matriculation number.
cout << "Name: " << s@name << endl;
cout << "Number: " << s@number << endl;
```

Because the relationship between Student and Person is applied automatically on demand, the subexpression s@name is actually equivalent to s@Person@name. Furthermore, if a function expects a Person argument p, it can be called with a Student object s, too, which will be implicitly converted to its associated Person object s@Person, i. e., Student objects can be used polymorphically as Person objects.

If a student object is accidentally created without an associated person object, an implicit conversion to Person – and, consequently, accessing any person attributes – would yield null. In practice, this can be avoided by always calling the above user-defined constructor instead of an attribute-initialization constructor and mutators to create a student. By declaring attributes in a private or protected section of a module (cf. Sec. 8.2), it is even possible to completely prevent such undesired calls outside the module.

Of course, it is also possible to define another Student constructor accepting a given name and last name instead of a pre-built person object in order to hide the explicit creation of this "subobject:"

```
// Create student with given name g, last name n,
// and matriculation number m.
Student (string g, string n, integer m) {
  return Student(Person(g, n), m);
}
```

By using this constructor, the fact that a student object actually consists of two associated subobjects (cf. Fig. 2) remains completely hidden. However, for reasons explained below (Sec. 6.4), it is always recommended to provide a constructor accepting a pre-built object of its base type, too.

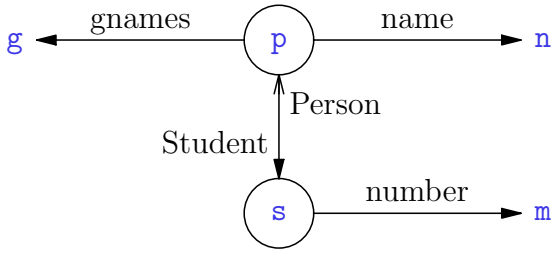


Figure 2: Student object s with associated person subobject p

The fact that the relationship between Student and Person is bidirectional can be exploited to check whether a given Person object "is" actually a student, i. e., to perform a *dynamic type test*, and to access its student attributes if appropriate, i. e., to perform a *downcast* (cf. Fig. 2):

```
// Polymorphically use a student as a person.
```

```
Person p = Student("Peter", "Clark", 777);

// Check whether p is actually a student s ...
if (Student s = p@Student) {
  // ... and access its matriculation number.
  cout << "Number: " << s@number << endl;
}
```

This corresponds roughly to a `dynamic_cast` in C++ which returns a valid pointer to an object of a derived class if the cast has been successful and a null pointer otherwise.

By employing automatic relationships to model object-oriented type hierarchies, the traditionally distinct or even conflicting concepts of *aggregation* (expressed by normal attributes and relationships) and *inheritance* (expressed by automatic attributes and relationships) have been merged into a single coherent concept. Furthermore, the fact that relationships can be defined incrementally, allows "supertypes" of a type to be declared later on, e. g.:

```
// Declare Vehicle as a "supertype" of Car.
typename Vehicle;
Car <->! Vehicle;
```

Despite its practical usefulness, such a possibility is missing in most statically typed object-oriented programming languages.

Multiple Inheritance

Of course, it is possible to use automatic relationships to model type hierarchies with multiple inheritance, too. For example, one might define a type `EmployedStudent` that is derived from both `Student` and another type `Employee`:

```
// Declare Employee as a subtype of Person.
typename Employee;
Employee <->! Person;

// Attributes and constructors of Employee.
Employee -> string company;
Employee (Person p, string c) { ..... }

// Declare EmployedStudent as a subtype of Student and Employee.
typename EmployedStudent;
EmployedStudent <->! Student;
EmployedStudent <->! Employee;
```

Since both of these types in turn "inherit" from `Person`, the typical question arises whether an `EmployedStudent` object should possess one or two `Person` subobjects, i. e., whether `Person` is, in C++ terminology, a *virtual* base type or not. In C++, the corresponding decision must be taken when the types `Student` and `Employee` are defined, even though it does not make any difference for *these* types. Therefore,

it would be much more logical to answer the question when `EmployedStudent` is defined, because only for this type (and possible subtypes of it) the distinction is relevant. However, the concept of automatic relationships does not provide a way to specify the difference at the level of *declarations*: The four `<->!` relationships between the types `Person`, `Student`, `Employee`, and `EmployedStudent` merely specify that there are two ways to convert an `EmployedStudent` to a `Person`, either via `Student` or via `Employee`, but they do not specify whether these ways lead to the same destination, i. e., to the same `Person` object, or not. Even though this appears to be disadvantageous at first sight, it will turn out to be the most flexible approach possible.

To actually distinguish between virtual and non-virtual inheritance, one simply creates either one or two `Person` "subobjects" when creating an `EmployedStudent` object in a constructor, e. g.:

```
// Create employed student with given name g, name n,
// matriculation number m, and company c.
EmployedStudent (string g, string n, integer m, string c) {
  Person p = Person(g, n);
  Student s = Student(p, m);
  Employee e = Employee(p, c);
  EmployedStudent es = EmployedStudent(@Student, s)(@Employee, e);
  return es;
}
```

Here, a *single* `Person` object `p` is created (by calling the user-defined `Person` constructor defined in Sec. 4.3), that is passed to both the `Student` and `Employee` constructors to create objects `s` and `e`, respectively, which *share* the subobject `p`. Afterwards, an `EmployedStudent` object `es` with subobjects `s` and `e` is created and returned (cf. Fig. 3). Therefore, converting an `EmployedStudent` object created by this constructor to type `Person` always yields the same `Person` subobject, no matter whether the conversion is done via `Student` or via `Employee`. However, since the compiler cannot know how a particular `EmployedStudent` object has been created, its conversion to `Person` is nevertheless *syntactically* ambiguous, and Sec. 6.5 discusses possibilities to resolve this.

The above example demonstrates the typical structure of a user-defined constructor, which calls (typically user-defined) constructors of its base types to construct the necessary subobjects and finally calls the predefined attribute-initialization constructor (plus mutators) of its own type to assemble the complete object.

Even though this parallels the structure of class constructors in C++ (and other languages), where a constructor of a derived type is forced to (explicitly or implicitly) call constructors of its base types, it is actually more flexible since the rules for calling these constructors – especially when virtual base classes are involved – are not hard-wired in the language specification, but can be freely defined by the programmer. In particular, it is easily possible to combine virtual and non-virtual inheritance

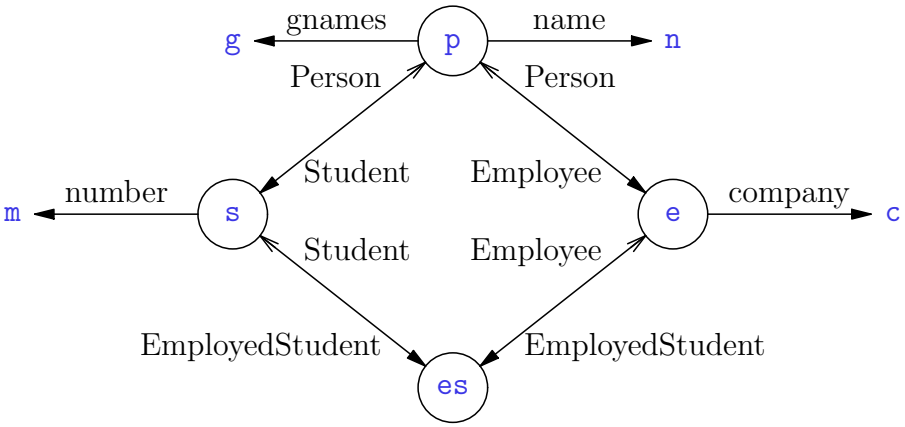


Figure 3: Employed student with shared person subobject

to model types such as F shown in Fig. 4 which inherits from four base types B, C, D, and E, which in turn inherit from two partially shared "incarnations" of A. Modelling such a structure in C++ would require artificial intermediate classes A1 and A2 derived non-virtually from A acting as virtual base classes of B and C resp. D and E.

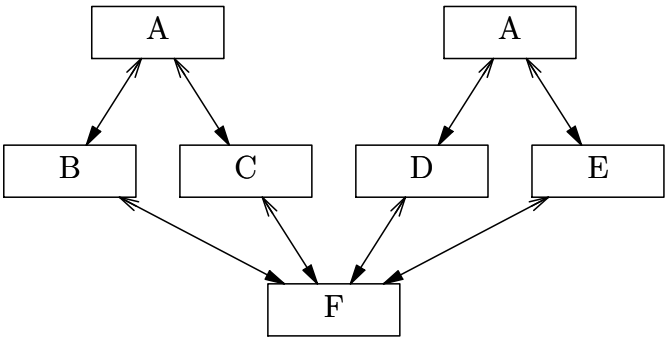


Figure 4: Combination of virtual and non-virtual inheritance

Apart from such rather esoteric examples, the approach to distinguish virtual from non-virtual inheritance in constructor *implementations* instead of in *declarations* appears to be more direct and clear than the C++ approach where the constructor of a virtual base class is only called from the constructor of the *most derived class* [33], even though constructors of intermediate classes may contain calls to it, too, which are simply ignored.

On the other hand, it might be argued that "hiding" the precise semantics of an inheritance hierarchy in constructor implementations does not contribute to comprehensibility either. Therefore, it is necessary to clearly document the intended semantics, e. g., in comments if there should be any possibility of uncertainty. Furthermore, the way of resolving the syntactically ambiguous conversion from EmployedStudent to Person that will be described in Sec. 6.5 will also help to make the intended semantics clear, even though it is still not strictly unambiguous.

On the other hand, by postponing the decision between virtual and non-virtual inheritance to constructor implementations, it would in fact be possible to define different constructors of the same type with different semantics. For example, one might define another constructor of `EmployedStudent` that creates a "schizo" whose first personality is a student (with given name `g1`, name `n1`, and matriculation number `m`) while its second personality is an employee (with given name `g2`, name `n2`, and company `c`), even though this is not expected to be very useful in practice (cf. Fig. 5):

```
// Create "schizo" employed student.
EmployedStudent (string g1, string n1, integer m,
                 string g2, string n2, string c) {
  Person p1 = Person(g1, n1); Student s = Student(p1, m);
  Person p2 = Person(g2, n2); Employee e = Employee(p2, c);
  EmployedStudent es = EmployedStudent(@Student, s)(@Employee, e);
  return es;
}
```

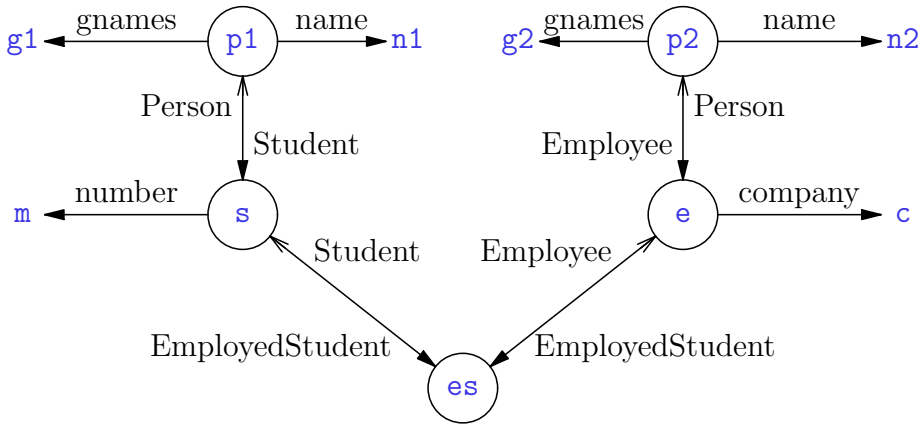


Figure 5: Employed student with two person subobjects

The example constructors of `EmployedStudent` shown in this subsection have demonstrated the usefulness of having constructors of base types (`Student` and `Employee` in these examples) which in turn accept pre-built objects of their base type(s) (here `Person`) as arguments. If, for example, the constructors `Student (Person, integer)` and `Employee (Person, string)` would not have been provided, and the predefined attribute-initialization constructors of `Student` and `Employee` can or should not be used due to information hiding concerns (cf. Sec. 8.2), it would be actually impossible to implement the constructor `EmployedStudent (string, string, integer, string)` shown earlier. This is because the other constructors of `Student` and `Employee` each create a separate `Person` object, so using them to create an `EmployedStudent` would actually create a "schizo," i. e., the "schizo constructor" shown above could also be implemented as follows:

```
// Create "schizo" employed student.
```

```
EmployedStudent (string g1, string n1, integer m,
                 string g2, string n2, string c) {
  Student s = Student(g1, n1, m);
  Employee e = Employee(g2, n2, c);
  return EmployedStudent(@Student, s)(@Employee, e);
}
```

Consequently, since `EmployedStudent` might also serve as a base type for other types, it is advisable to provide an additional constructor for it that accepts pre-built objects of its (direct) base types, too:

```
// Create employed student with
// student part s and employee part e.
EmployedStudent (Student s, Employee e) {
  return EmployedStudent(@Student, s)(@Employee, e);
}
```

Conflict Resolution

The fact that automatic attributes and relationships are actually just attributes and relationships implies that, e. g., a `Student` object does not really *inherit* anything from its associated `Person` object, but rather simply *refers* to the latter, while the apparent "inheritance" simply results from the automatic application of the corresponding attribute. By that means, the full advantages of object-oriented subtype polymorphism – including multiple and repeated inheritance – are available, while at the same time avoiding many problems usually associated with these mechanisms in other languages.

In particular, *name conflicts* between multiply "inherited" attributes are simply resolved by qualifying attribute names with the name of their original type, e. g.:

```
// Personal number of employee.
Employee -> integer number;

// Inspect matriculation number and
// personal number of employed student.
EmployedStudent es = ......;
cout << es@Student@number << endl;
cout << es@Employee@number << endl;

// Change these numbers.
es(@Student@number, ...)(@Employee@number, ...);
```

While the inspection of such attributes with the `@` operator is straightforward, since `es` actually possesses (anonymous) attributes named `Student` and `Employee` and the objects `es@Student` and `es@Employee` both possess (different) attributes named `number`, a mutator call such as `es(@Student@number, ...)` is actually interpreted as `es@Student(@number, ...)`. The only difference is that the former returns the

original object `es` while the latter would return the intermediate object `es@Student`.

If the type `EmployedStudent` itself would possess another attribute `number`, this could be directly accessed with its unqualified name, e. g., `es@number`, since in that case no implicit conversion of the object `es` is necessary at all.

Another potential source of conflicts are repeatedly "inherited" types. Rephrased in open types terminology, such a conflict arises when the graph induced by automatic attribute declarations contains multiple paths from a source type (such as `EmployedStudent`) to a target type (such as `Person`). In such a case, trying to use an `EmployedStudent` object `es` as a `Person` object is ambiguous since it could mean either `es@Student@Person` or `es@Employee@Person`. As the (of course rather artificial) example of a schizophrenic employed student demonstrated, both interpretations might actually lead to different results in practice, even though under "normal" circumstances they are expected to yield the same person object. While in C++, these "normal" circumstances can be expressed by declaring `Person` as a virtual base type of `Student` and `Employee`, this information has not been made explicit with open types yet. Therefore, trying to assign `es` to a `Person` variable or trying to directly access a person attribute such as `es@name` actually leads to a compile time error due to ambiguity, and one has to write, e. g., `es@Student@Person@name`, or just `es@Student@name` since the intermediate student object `es@Student` is automatically convertible to its associated person object `es@Student@Person`.

To avoid this significant inconvenience and to explicitly declare that an `EmployedStudent` shall possess only one `Person` subobject, an additional *direct* relationship between these types can be declared:

```
EmployedStudent <->! Person;
```

At first sight, this does not seem to solve the problem, but rather to aggravate it, since it introduces a third path from `EmployedStudent` to `Person`. However, a *direct* conversion path between two types is generally preferred over any indirect path. Therefore, using an employed student object `es` where a person object is expected is now unambiguously interpreted as `es@Person`, not as `es@Student@Person` or `es@Employee@Person`. In particular, person attributes such as `name` might now be used directly, e. g., `es@name` (meaning `es@Person@name`) or `es(@name, ...)` (meaning `es(@Person@name, ...)`).

The final step to make this actually work as expected, however, is to change all `EmployedStudent` constructors to explicitly initialize the new `Person` attribute, e. g. (cf. Fig. 6):

```
// Create employed student with given name g, name n,
// matriculation number m, and company c.
EmployedStudent (string g, string n, integer m, string c) {
  Person p = Person(g, n);
  Student s = Student(p, m);
  Employee e = Employee(p, c);
  EmployedStudent es = EmployedStudent(@Person, p)
```

```
    (@Student, s)(@Employee, e);
  return es;
}
```

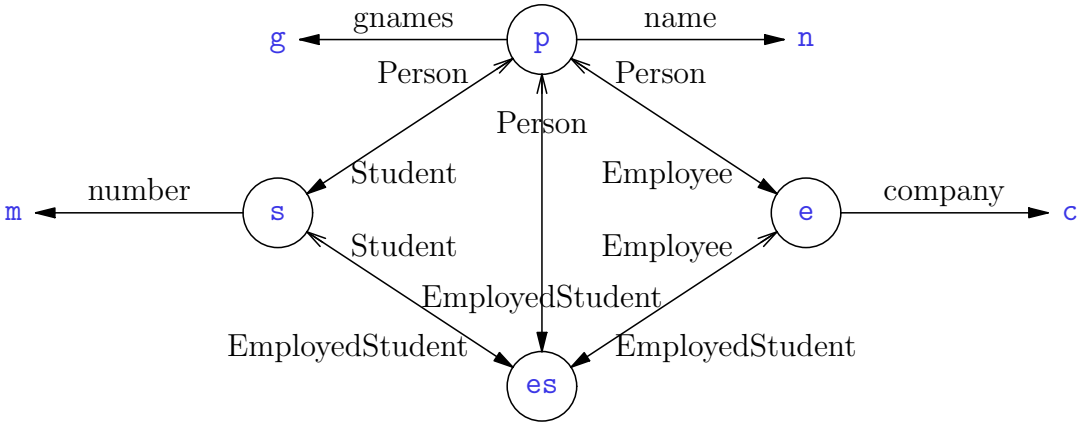


Figure 6: Employed student with direct relationship to person

Since now all three paths from an `EmployedStudent` object to `Person` lead to the same person subobject, this object might also be accessed indirectly via the `Student` or `Employee` subobjects.

From a C++ point of view, declaring the direct automatic relationship between `EmployedStudent` and `Person` (and appropriately implementing the `EmployedStudent` constructors) achieves the same effect as declaring `Person` as a virtual base class of `Student` and `Employee`. However, declaring this in the context of `EmployedStudent` is much more logical, since for `Student` and `Employee` themselves the distinction between virtual and non-virtual inheritance from `Person` is completely irrelevant.

On the other hand, the bare declaration of this relationship does not automatically force programmers to appropriately implement the necessary constructors: In principle, it would still be possible to implement a "schizo constructor" as shown earlier which may not even initialize the direct relationship between `EmployedStudent` and `Person` – or initialize it arbitrarily to one of the schizo's personalities or even to a third personality! Since the programmer has complete freedom, he also has increased possibilities to make nonsense, of course.

Repeated Inheritance

By using one-to-many relationships, it is also possible to model repeated inheritance, e. g., a student with multiple enrollments:

```
// Multiply enrolled student.
typename MultiStudent;
```

```
MultiStudent <->> Student StudParts;

// Create student with two enrollments.
Person p = Person("Peter", "Clark");
MultiStudent ms = MultiStudent(@StudParts, Student(p, 123))
 (@StudParts, Student(p, 456));

// Polymorphic use of both student parts.
Student s1 = ms@StudParts[1];
Student s2 = ms@StudParts[2];
```

Dynamic Object Evolution

The fact that an object of a derived type such as `Student` or `EmployedStudent` is actually a network of interconnected subobjects – even though this remains invisible except when constructing the objects –, can be exploited in a straightforward manner to implement *dynamic object evolution* in a completely type-safe manner. For example, it is almost trivial to transform an object that has been initially created as a bare person into a student, an employee, or even an employed student later, by simply creating additional associated subobjects, e. g.:

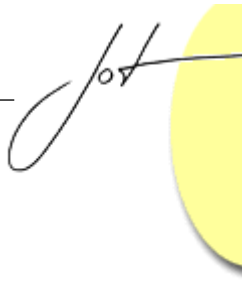
```
// Create a person object p.
Person p = Person("Peter", "Clark");

// Check whether it is a student.
if (p@Student) ..... // Not yet.

// Create an alias q for p.
Person q = p;

// Check whether p and q refer to the same object.
if (p == q) .....    // Yes.

// "Transform" p to a student.
p(@Student, Student(@number, 777));

// Check whether it is a student.
if (p@Student) ..... // Yes!

// Check whether p and q still refer to the same object.
if (p == q) .....    // Yes!
```

Instead of directly using the mutator call `p(@Student, ...)` to transform `p` to a student, one might also pass `p` to the `Student` constructor accepting a pre-built `Person` object (and ignore its result):

```
// "Transform" p to a student.
Student(p, 777);
```

This is another strong argument for always providing these kinds of constructors.

Conversely, it is also possible to delete subobjects to transform a specialized



object to a more general one, e. g.:

```
// Obtain student reference to p.
Student s = p@Student;

// "Transform" p to a person by deleting its student subobject.
~ p@Student;

// Check whether it is a Student.
if (p@Student) ..... // No longer.

// Check whether s is still valid.
if (s) .....         // No!
```

Here, it does not matter whether `p` has been originally created as a `Person` or a `Student` (or something else).

By explicitly deleting the student subobject associated with `p`, all "student references" to this person (such as `s` in the example) automatically become null. Otherwise, if only the association between `p` and its student object would have been cut, these references would remain valid, but refer to a degenerate student object that does not possess an associated person object anymore.[8]

It is even possible to create "hybrid" objects, such as a person that is both a student and an employee, even if no common "subtype" of these types (such as `EmployedStudent`) would exist.

7 GLOBAL VIRTUAL FUNCTIONS

Basic Concept

So far, open types, attributes, and relationships have been described as an approach to model data structures in programming languages that is much more flexible than procedural record-based and object-oriented class-based data models. The essential reason for this increased flexibility is the fact that type definitions are completely separated from the definition of their "constituents," i. e, their data fields (modeled by attributes), associations (modeled by relationships), and base types (modeled by automatic attributes or relationships). Actually, an open type itself is just an "empty shell" whose "substance" is defined incrementally. (This parallels the philosophical observation that a *concept* is meaningless in itself, but gains its meaning only from its associations to other concepts.)

Based on this principle, it is logically consistent that an open type itself does not possess any *operations* (or "methods") either, but that operations on types are defined incrementally, too. This goes in line with traditional procedural programming languages where operations are defined as free-standing procedures or functions.

[8] According to the rules given in Sec. 4.5, access to this person object and its attributes would still be well-defined, however.

For example, the following global function concatenates all names of a particular person p:

```
string format (Person p) {
  string s = "";
  for (string g : p@gnames) s += g + " ";
  return s + g@name;
}
```

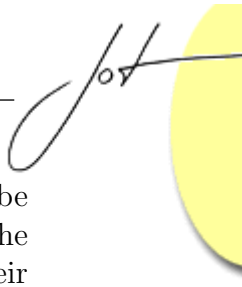
Since global functions are not bound to any type, it is always possible to add new functions in a *modular* way, i. e., without needing to touch any previously written source code. Even though this fact is almost too trivial to mention, it should be noted that methods or member functions of classes in statically typed languages usually do not exhibit this highly desirable property: To add a new method to a class, it is necessary to change and recompile the class definition as well as to recompile all code depending on it, which might be a considerable amount.

On the other hand, simple global functions such as the one shown above lack the also highly desirable property of being *overridable* or *redefinable*, that is provided by methods or virtual member functions. To reconcile these apparently conflicting properties of global functions and virtual member functions and to combine their advantages without encountering their disadvantages, *global virtual functions* (GVFs) have been proposed earlier [14, 16]. Even though the basic idea is very simple, its consequences are rather far-reaching: By declaring a global function `virtual` (which is not allowed in C++), it is possible to override or redefine it later by simply giving a new definition. (To use C++ terminology, this means that the *one definition rule*, that basically says that no program entity must be defined more than once, does not hold for global virtual functions.) However, every new definition (which is also called a *branch* of the function) is able to call the previous branch on demand by using the keyword `virtual` as the name of a pseudo-function referring to this branch.

For example, if the function `format` shown above would have been defined `virtual`, it could be redefined as follows:

```
virtual string format (Person p) {
  if (p) return virtual();
  else return "Null person";
}
```

Here, the redefinition checks whether `p` actually refers to a real person object before calling the previous branch to concatenate its names. Otherwise, i. e., if `p` is null, a more instructive text is returned than would be produced by the previous branch, which obviously did not take this case into account. It should be noted that the previous branch denoted by the keyword `virtual` is always called without explicit arguments, since the original arguments of the call are implicitly passed unchanged (even if the formal parameters would have been modified). Furthermore, if the first branch calls its previous branch, a predefined "branch zero" with an empty body is executed, which returns a null value of the function's result type (cf. Sec. 3).



If a function is redefined multiple times, the order of redefinitions *might* be important, even though in many practical cases it is actually irrelevant. If both the original definition and all redefinitions appear in the same translation unit, their textual order is decisive: A call to the function will always call the *last* branch, which might call the second-last as its previous branch, which in turn might call the third-last as its previous branch, and so on. If branches are distributed over multiple translation units, the concept of modules introduced in Sec. 8 determines their precise order (cf. Sec. 8.3). In any case, however, a simple linear chain of branches is built up.

Possible Applications

This simple *general* mechanism, which is hardly more complex than functions in C or procedures in other procedural languages, can be used to solve a broad range of problems for which many different *specialized* mechanisms are provided in object-oriented and aspect-oriented programming languages:

- A redefinition can execute additional prologue and/or epilogue code before/after calling the previous branch.
  This can be used, for example, to implement retroactive synchronization, monitoring, or profiling, to notify observers of an object, etc. It subsumes the concept of before/after/around methods in CLOS [21] and similar languages and that of before/after/around advice in aspect-oriented languages such as AspectC++ [32].

- A redefinition can execute completely different code, without calling the previous branch at all.
  This can be used, for example, to patch erroneous functions whose source code is not available, to provide a more efficient implementation for performance-critical functions, etc. It is also covered by around methods and advice.

- A redefinition can check some condition on its arguments (or any other data) and call the previous branch only if it is satisfied while signalling an error or exception otherwise.
  This can be used, for example, to retroactively check necessary preconditions to implement the "design by contract" model, to implement protection, etc.

- A redefinition can check some condition on its arguments (or any other data) and execute more specialized code if it is satisfied, while delegating the call to the previous branch otherwise.
  This can be used, for example, to treat unanticipated special cases differently, as in the example of `format` given above. Furthermore, it can be used to simulate the well-known *dynamic dispatch* found in object-oriented languages by checking whether the function's first argument is actually an instance of a subtype of its static type and executing specialized code in that case, while calling the previous, more general branch otherwise.

Since the condition might also check the dynamic types of multiple arguments – or any other predicate over them –, it is equally easy to simulate *multiple* and even *predicate-based* dispatch [9].

Since the last possibility – check a particular condition and execute the previous branch if it is not satisfied – occurs so frequently in practice, some syntactic sugar is provided to simplify its usage: By specifying a *guard* consisting of the keyword `if` and a subsequent condition in the function's head – after the parameter list and an optional `throw` clause, but before the function's body –, this branch is executed only if the condition is satisfied, while otherwise the call is delegated to the previous branch. For example, the redefinition of `format` shown above might be written more compactly as follows:

```
virtual string format (Person p) if (!p) {
  return "Null person";
}
```

Equivalence of Attributes and Global Virtual Functions

Even though attributes of open types and virtual functions[9] have been presented as separate concepts so far, they are actually interrelated as follows.

A single-valued attribute such as

```
Person -> string name;
```

is actually equivalent to a pair of virtual functions, one for getting and one for setting the attribute's value for a particular object:

```
// Get value of attribute name for person x.
virtual string name (Person x) { ..... }

// Set value of attribute name for person x to y and return y.
virtual string name (Person x, string y) { ..... }
```

These functions are called automatically, whenever an attribute inspection `x@name` or an attribute mutator `x(@name, y)` is executed, respectively. While the first branch of these functions is predefined and cannot be changed directly, it is possible to define additional branches for them which modify or extend the original behaviour, for example:

```
// Redefine get and set function of attribute name
// to perform logging.
virtual string name (Person x) {
  string y = virtual(); // Call original implementation.
  cout << "get name: " << y << endl; // Perform logging.
  return y;
```

[9] Global virtual functions are simply called virtual functions in the sequel, as there is no possibility of confusion with virtual member functions.

```
}
virtual string name (Person x, string y) {
  virtual();            // Call original implementation.
  cout << "set name: " << y << endl; // Perform logging.
  return y;
}
```

Now, attribute accesses `x@name` and `x(@name, y)` will call these redefinitions of the get resp. set function, which in turn call the original implementations via `virtual`.

Similarly, a multi-valued attribute is actually equivalent to three virtual functions, one for getting an object's sequence of attribute values, one for appending or inserting a value (in)to it, and one for removing a value from it. Finally, a bidirectional relationship is equivalent to a pair of attributes, i. e., to four (for 1:1 relationships), five (1:N and N:1), or six (N:N) virtual functions: a get function for each direction, plus one or two update functions (i. e., a set function or an insert and a remove function) for each direction. Since both directions of the relationship shall be kept consistent automatically, each of the update functions calls the appropriate function of the other direction with reversed arguments after it has performed its own update operation. To prevent infinite mutual recursion, however, the other function is only called if the current function has not itself been called from it.

The fact that all these functions defining attributes and relationships can be redefined, significantly increases the flexibility of these concepts once more and opens the door to many advanced applications such as implementations of the *Observer Pattern* [10] without any preparation or cooperation of the observed objects, transparent implementation of persistent attributes, implementation of derived attributes, etc.

Due to the equivalence of attributes and virtual functions, the attribute inspection operator `@` as well as attribute initialization constructors and mutators are actually just syntactic sugar supporting a more object-oriented instead of a functional or procedural interface to open type objects.

Virtual Constructors

Since user-defined constructors of open types are actually global functions (cf. Sec. 4.3), it is possible to define them `virtual`, too, which also increases their flexibility. Furthermore, the predefined empty constructor of an open type is actually a virtual constructor, which can be redefined on demand, e. g.:[10]

```
// Original definition of Car.
typename Car;

// Later definition of Vehicle as a "supertype" of Car.
typename Vehicle;
```

[10] Just like a call to the empty constructor requires a pseudo-argument `@` to distinguish it from a call to the parameterless constructor (cf. Sec. 4.3), its definition also requires a pseudo-parameter declaration `@`.

```
Car <->! Vehicle;

// Redefinition of empty Car constructor
// creating an associated Vehicle object.
virtual Car (@) {
  Car c = virtual(); // Call original constructor to create a car.
  c(@Vehicle, Vehicle(@)); // Create an associated vehicle object.
  return c;                // Return the car.
}
```

Since the empty constructor is the only way to actually create a new object, which is called directly by each attribute-initialization constructor and either directly or indirectly by each user-defined constructor, a redefinition of the former is a reliable way to intercept any object creation and to associate extra code with it.

8 MODULES

Motivation

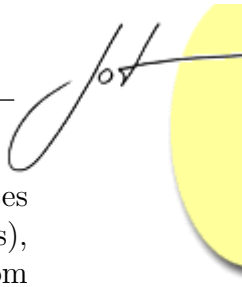
If types, their attributes and relationships, and the functions operating on them are all global entities, it is hard if not impossible to write *modular software* and to enforce the principle of *information hiding* [31].

To achieve some basic structuring of a large system into more comprehensible units and to reduce the danger of accidental name clashes between globally defined entities, C++ provides the concept of *namespaces* [33]. If this is combined with the possibility provided by classes to divide them into public, protected, and private sections, a simple module concept similar to that of Oberon [37] is achieved. In particular, information hiding becomes possible if the public part of a namespace – which is also called a *module* now – contains only definitions of types, attributes, and functions which should be visible to other modules (i. e., the externally visible *interface* of the module), while internal entities are defined in the private part. It is even possible to reduce functions specified in the public part to bare *declarations* without a body and to give the full definitions only in the private part.

Protected Definitions

While all entities declared in a private section of a module are completely invisible to other modules, those defined in a public section are fully accessible outside the module. In particular, a public open type is provided together with its predefined constructors (cf. Sec. 4.3), a public attribute or relationship can be both inspected and updated from within other modules, and a public virtual function can be both called and redefined elsewhere.

However, when taking the information hiding principle seriously, it should be possible to separate these issues, i. e., it should be possible to provide open types without their empty constructors (to force clients to use other constructors since



completely empty objects might be semantically meaningless), to provide attributes and relationships read-only (since arbitrary updates might violate object invariants), and to provide virtual functions which can only be redefined, but not called from other modules (which might appear strange at first sight, but will be explained below). To achieve all these aims, the corresponding entities can be defined in a `protected` section of a module.

If an open type is provided that way, only its parameterless constructor returning a null object/reference as well as its attribute-initialization constructor and mutator can be used outside the defining module. The latter are needed to allow other modules to define attributes for the type and to initialize resp. modify their values. However, to enforce the information hiding principle, they can only be called with attributes whose update function(s) are accessible (cf. below).

If an attribute or relationships is provided in a protected section, only its get function(s) can be used outside the defining module, i. e., it can only be inspected using the `@` operator, but neither initialized nor modified using an attribute-initialization constructor or mutator.

Finally, if a virtual function is provided in a protected section, it can only be redefined, but not called from outside the defining module. This can be used, for example, to provide "hooks" to internal functions of a module at which other modules can "hang" on extensions (by redefining the functions in a way that always calls the previous branch). The other combination of "access rights" to a GVF, i. e., that it can only be called, but not redefined outside its defining module, can be achieved in principle by declaring it as an ordinary, non-virtual function in the module's public part and implementing it as a virtual function in the private part. However, doing so actually leads the basic idea of GVFs ad absurdum: If functions cannot be redefined outside their defining module, it would be necessary to modify and recompile this module whenever (anticipated or unanticipated) changes or extensions to one of its functions become necessary. Furthermore, inside a single module the possibility to redefine a function is only of limited usefulness.

Inter-Module Dependencies

In addition to information hiding concerns, another important purpose of modules is to define a precise *initialization order* for programs consisting of multiple translation units.

In standard C++, the order in which global variables of different translation units are initialized at program startup is completely undefined. In particular, if a function defined in some translation unit uses the value of a global variable defined and initialized in the same translation unit, it is not guaranteed that the variable is properly initialized before the function gets called. Even though it is possible to circumvent this special problem with a trick (using static variables in functions which *are* guaranteed to get initialized before they are used), this situation is generally unsatisfactory. Furthermore, if different branches of the same virtual function are

distributed over multiple translation units, their *activation order* is important to precisely specify which branch is (currently) the *last* branch of the function (cf. Sec. 7.1).

To meet these requirements, a module might be specified to *depend* on one or more other modules by listing the names of these "base modules" in a way similar to the base classes of a class, e. g.:

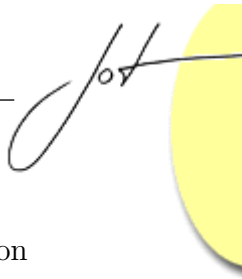
```
namespace A {
  virtual void f () { ..... }      // 1
}
namespace B : A {
  virtual void A::f () { ..... }   // 3
  virtual void A::f () { ..... }   // 4
}
namespace C : A {
  virtual void A::f () { ..... }   // 2
}
namespace D : C, B {
  ......
}
```

Here, module `D` depends on modules `C` and `B` which both in turn depend on module `A`. Therefore, at program startup time, `C` and `B` will be initialized – in this order – before `D`, while `A` will be initialized (once) before `B` and `C`, i. e., the overall initialization order will be `A`, `C`, `B`, `D`. This implies in turn that the branches of the function `A::f`[11] will be activated in the order indicated by the comments above.

Formally, the module initialization order of a program is obtained by traversing the directed acyclic graph consisting of modules (nodes) and inter-module dependencies (edges) in a depth-first, left-to-right manner (where left-to-right corresponds to the textual order in which base modules of a module are specified), starting at a designated main module and visiting each module exactly once (i. e., ignoring already visited ones).

Initializing a single module means to initialize the global variables of the module by executing their initializer expressions and/or calling appropriate constructors and to activate the branches of virtual functions defined in the module, where the order of these steps is given by the textual order of the corresponding definitions. In particular, variable initializations and branch activations will be performed interleaved to guarantee on the one hand that all variables defined before a particular branch are initialized prior to any execution of this branch and on the other hand that all branches defined before an initializer expression are activated when this expression is evaluated. Together with the general C++ rule of "declare before use"

[11] Note that to redefine a function of a foreign module it is necessary to qualify its name with the name of this module, since an unqualified name would denote a different function introduced in the current module.



this guarantees that all variables are properly initialized before their first use.

If, for example, a module defines the following variables and virtual function branches:

```
integer x = 1;
virtual integer f () { return x + 1; }
integer y = f();
virtual integer f () { return y + 1; }
integer z = f();
```

these are initialized resp. activated as follows: First, variable `x` is initialized to `1`. Then, the first branch of the virtual function `f` is activated causing it to become the currently last branch of this function. Next, variable `y` is initialized by calling this function, i. e., by executing exactly this branch, resulting in the value `2`. Then, the second branch of `f` is activated causing it to become the last branch of the function now. Finally, variable `z` is initialized by calling this function, too, i. e., by executing this new branch, resulting in the value `3`. The important thing to notice in this example is that the call to `f` in the initialization of `y` does *not* call the last branch of the function defined in the whole module – which depends on the value of exactly this variable –, but rather the last branch activated *so far*.

To complete these details of module initializations, it should be noted that there is also a *deactivation order* of branches which is interleaved with the "de-initialization" of global variables, i. e., the execution of their destructors. The latter are executed in exactly the reverse order of the corresponding constructor calls.

Deactivating a branch of a virtual function means to remove it from the linear chain of branches of this function, causing its previous branch to become the currently last branch. This is again important if a branch depends on the values of global variables defined earlier: As soon as a destructor has been executed for such a variable, its value is undefined, and therefore the branches defined after it should no longer get executed.

Type-Safe Separate Compilation

Besides promoting information hiding and establishing a precise global initialization order, modules support the decomposition of a large program into smaller parts which can be compiled separately without sacrificing static type safety.

If some module `B` wants to use an entity `x` defined in another module `A`, the latter must be specified as a base module of the former. Afterwards, according to the normal rules for name lookup in C++, the entity might be accessed either via its qualified name `A::x` or via the unqualified name `x` if it has been "imported" by a declaration `using A::x;` To simplify the import of many entities from the same module, an extended `using` declaration such as `using A { x, y, z };` is provided.

Since each module constitutes a separate C++ namespace, name clashes between modules are excluded by definition. In particular, if both `A` and `B` define an open

type `T`, no matter whether it is public, protected, or private, this yields two completely different types `A::T` and `B::T`. If both shall be used in a third module `C`, at most one of them must be imported by a `using` declaration to avoid ambiguity. Similarly, if both `A` and `B` define an attribute `a` for the same type `U`, this yields unrelated attributes `A::a` and `B::a`, which can be used simultaneously if necessary, e. g., as `U(@A::a, ...)(@B::a, ...)`. Therefore, it is in fact possible to compile and completely type-check modules separately, without any danger of encountering type errors or name clashes at link or run time.

This is in sharp contrast to, e. g., AspectJ [22], where multiple extensions of the same class might in fact conflict with each other, leading to errors at weave time, even if all aspects have been compiled successfully in isolation.

Of course, to type-check the usage of an entity defined in a different module, the (pre)compiler must consult its definition in the public or protected part of this module. This implies that a module must be recompiled if its own source code and/or the non-private part of one of its base modules has been modified since the last compilation. To release the programmer from the burden of tracking these dependencies manually, the C++ precompiler checks the timestamps of all base modules and automatically performs the necessary recompilations similar to the Unix `make` facility.

Dynamically Loaded Modules

Since the process of linking modules together, which have been compiled and type-checked separately, cannot lead to any errors, it is even possible to delay this process partially or completely until run time, i. e., to load (and unload) modules *dynamically*, similar to the way Java classes are loaded dynamically. Here, loading a module simply means to initialize it according to the rules given in Sec. 8.3, i. e., to initialize its global variables and to activate the GVF branches defined in the module. To make sure that everything the module depends on is actually available, its direct and indirect base modules are automatically loaded/initialized before in the order described in Sec. 8.3, too. In particular, modules which have already been initialized, either at program start time or during an earlier dynamic load operation, will not be initialized again.

Typically, but not necessarily, dynamically loaded modules do not introduce any new (public) functions into a program, but only provide additional branches for already existing virtual functions. Since these branches will be activated when the module containing their definitions gets loaded, they become *implicitly* accessible via the GVF they belong to. Of course, such a dynamically loaded branch of a GVF might internally use other functions and global variables defined in its module, which will not be directly accessible to the remaining program. Furthermore, a dynamically loaded module might contain declarations of additional attributes for open types defined in other modules. Again, these attributes will neither be known to nor directly accessible by other modules of the program, but the functions and GVF branches defined in the same module can use them.

9 THE EXPRESSION PROBLEM

To complete the conceptual part of the paper, this section presents a coherent example, namely a solution to the well-known *expression problem* [35], that demonstrates a typical application of open types, global virtual functions, and modules and their use in concert. In particular, it illustrates their ability to perform *non-invasive* extensions and modifications of an existing software system, which is particularly useful to cope with *unanticipated software evolution*.

As a starting point, Fig. 7 shows a module `expr` providing a basic implementation of *arithmetic expressions*, consisting of:

- a protected open type `Expr` with five protected attributes, resembling a variant record as described in Sec. 4.7;
- three public virtual constructors of this type to construct constant, unary, and binary expressions, respectively;
- a public virtual function `eval` to evaluate an expression, i. e., to compute its value, with three branches corresponding to these different kinds of expressions.

According to Sec. 8.2, declaring `Expr` and its attributes protected forces client modules to use the public user-defined constructors to create objects, and these objects will be immutable since no functions performing attribute update operations are provided.

Based on this module, Fig. 8 shows an operational extension that provides another public[12] virtual function `print` to print an expression on the standard output stream `cout`. This is a typical example of a dynamically bound function that can be added to a system in a completely modular way, i. e., without touching or recompiling any existing source code.

The following module `rem` retroactively extends the function `eval` defined in module `expr` with a branch that evaluates remainder expressions denoted by the `%` operator (since the constructor for binary expressions defined in module `expr` and the print function defined in module `print` are already general enough to properly handle such expressions, they do not need to be extended):

```
namespace rem : expr {
  using expr { Expr };

  // Evaluate remainder expressions.
  virtual integer expr::eval (Expr x) if (x@op == '%') {
    return eval(x@left) % eval(x@right);
  }
}
```

[12] The first section of a module is implicitly public.

```
namespace expr {
protected:
  typename Expr;         // Arithmetic expression.
  Expr -> integer val;   // Value of a constant expression.
  Expr -> character op; // Operator of a compound expression.
  Expr -> Expr body;     // Body of a unary expression.
  Expr -> Expr left;     // Left and right operand
  Expr -> Expr right;    // of a binary expression.

public:
  // Create constant expression with value v.
  virtual Expr (integer v) { return Expr(@val, v); }

  // Create unary expression with operator o and body b.
  virtual Expr (character o, Expr b) {
    return Expr(@op, o)(@body, b);
  }

  // Create binary expr. with operator o and operands l and r.
  virtual Expr (Expr l, character o, Expr r) {
    return Expr(@op, o)(@left, l)(@right, r);
  }

  // Evaluate constant expression.
  virtual integer eval (Expr x) if (x@val) {
    return x@val;
  }

  // Evaluate unary expressions.
  virtual integer eval (Expr x) if (x@body) {
    if (x@op == '+') return eval(x@body);
    if (x@op == '-') return -eval(x@body);
  }

  // Evaluate binary expressions.
  virtual integer eval (Expr x) if (x@left) {
    if (x@op == '+') return eval(x@left) + eval(x@right);
    if (x@op == '-') return eval(x@left) - eval(x@right);
    if (x@op == '*') return eval(x@left) * eval(x@right);
    if (x@op == '/') return eval(x@left) / eval(x@right);
  }
}
```

Figure 7: Basic implementation of arithmetic expressions

```
namespace print : iostream, expr {
  using iostream { cout };
  using expr { Expr };

  // Print constant expression.
  virtual void print (Expr x) if (x@val) {
    cout << x@val;
  }

  // Print unary expression.
  virtual void print (Expr x) if (x@body) {
    cout << x@op << '('; print(x@body); cout << ')';
  }

  // Print binary expression.
  virtual void print (Expr x) if (x@left) {
    cout << '('; print(x@left); cout << ')';
    cout << x@op;
    cout << '('; print(x@right); cout << ')';
  }
}
```

Figure 8: An operational extension

Again, this extension can be performed in a completely modular way.

Finally, the following module `cache` provides a redefinition of `expr::eval` that calls its previous branch via the pseudo-function `virtual` exactly once per expression `x` and caches its result value in a private attribute `val` (which is completely distinct from the attribute of the same name defined in module `expr`, cf. Sec. 8.4):

```
namespace cache : expr {
  using expr { Expr };
private:
  // Cached value of expression.
  Expr -> integer val;

  // Evaluate expression x and cache its value.
  virtual integer expr::eval (Expr x) {
    if (!x@val) x(@val, virtual());
    return x@val;
  }
}
```

Since this attribute is declared `private`, it is completely inaccessible, i. e., virtually invisible, to any other module. This is indeed appropriate when taking the information hiding principle seriously, since the branch of `expr::eval` defined in this module is the only function that needs to know about this attribute and requires

access to it.[13] Since this module does not introduce any new functions, it would even be possible to load it dynamically on demand to enable caching of expression values at run time.

To complete the example, the following module `main` assembles the components defined by the previous modules and provides a global statement block that replaces the traditional `main` function of C++ programs:

```
// Statically link modules expr, rem, and print.
namespace main : expr, rem, print {
  using expr { Expr, eval };
  using print { print };

  // Global statement block constituting the main program.
  {
    // Dynamically load module cache in some cases.
    if (......) load("cache");

    // Create, evaluate, and print expressions.
    Expr x = Expr(......);
    integer v = eval(x);
    print(x);
    ......
  }
}
```

10 IMPLEMENTATION

Possible approaches to implement types with null values have been described in [15], while the implementation of global virtual functions and modules has been described in [14, 16]. Therefore, the remainder of this section deals with the implementation of open types and attributes only. Here, the particular challenge is to find a way to store objects compactly, especially to avoid storage consumption for unused attributes, without causing attribute access operations to become overly expensive, i. e., to provide a reasonable balance between run time efficiency and storage consumption. The possibility to load attributes of a type dynamically, even after objects of the type have been created, further complicates these matters.

Basic Attribute Management

Figure 9 illustrates the basic approach to implement open type objects possessing values for different subsets of attributes. Here, an open type value is actually a pointer to an *intermediate cell*, which in turn references an associated *data area*

[13] Since a redefinition of a global virtual function is never called directly, it does not make any difference whether it is declared public, protected, or private.

containing the object's attribute values (in decreasing order of their alignment restrictions to avoid any padding in between) and a shared *object descriptor* describing its contents. For each attribute of the open type (`Expr` in this example), an object descriptor contains the information whether and where the corresponding attribute value is stored in an object's data area: If the "slot" corresponding to the attribute contains a non-negative number, it denotes the relative address of the attribute's value in the data area, while a negative value or dash indicates the absence of the attribute.

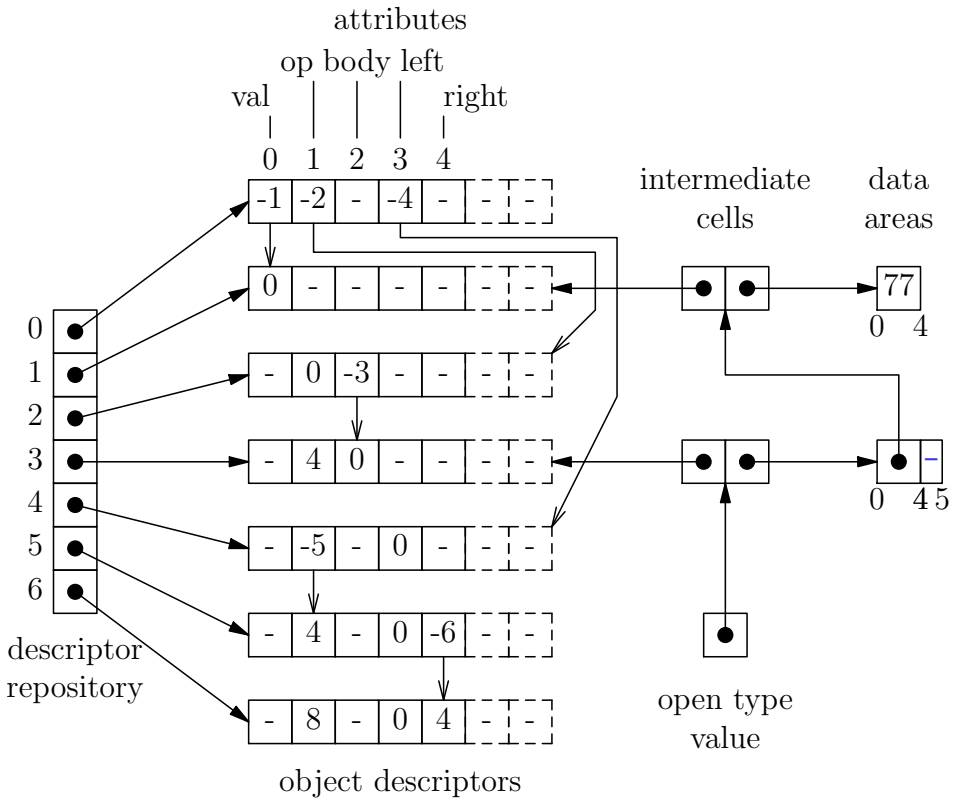


Figure 9: Basic attribute management

Based on this information, an attribute access operation proceeds as follows: A unique ordinal number assigned to the attribute (e. g., `1` for `op`) is used to access the corresponding slot of the object's descriptor, and the offset value found there (non-negative number) is used to access the attribute's value in the object's data area. If no offset value is found, however (negative number or dash), reading the attribute simply returns the null value of its target type, while a write access must extend the object's data area in order to obtain space for the new attribute. (This might change the address of the data area, but the address of the intermediate cell stored in open type values remains stable.) Furthermore, the object's descriptor pointer must be redirected to the "successor" descriptor that contains exactly the same attributes as the current one plus the new attribute.

Since object descriptors are created dynamically on demand, this extended de-

scriptor might not yet exist. In that case, it is created and inserted into the type's *descriptor repository*. Furthermore, it is compared with all currently existing descriptors in order to find its "successors," i. e., those descriptors containing the same subset of attributes plus one extra attribute, as well as its "predecessors," i. e., those descriptors for which the new one is a successor. These predecessor/successor relationships are remembered by replacing the dash in the predecessor's slot corresponding to the extra attribute with a logical reference to the successor (depicted by an arrow in Fig. 9), i. e., with the negated value of its index in the descriptor repository. Using this information, an already existing successor descriptor can be found easily and efficiently via the descriptor repository.

Typically, an object has to be extended a few times during its initialization, while afterwards the set of its attributes (but, of course, not necessarily their values) is expected to remain rather stable. Similarly, the descriptor "network" of an open type is expected to change a number of times during the initialization phase of a program, until the object descriptors and their predecessor/successor relationships required for the typical object initialization patterns of the program have been built up. Afterwards, most object descriptors needed to perform an object extension will be found immediately via the negated index in the corresponding slot of the object's current descriptor.

Since object descriptors are only created on demand, their number usually remains rather small compared to their theoretically possible exponential number. For example, Fig. 9 shows 7 out of 32 possible descriptors which are required by the three user-defined `Expr` constructors shown in Sec. 9.

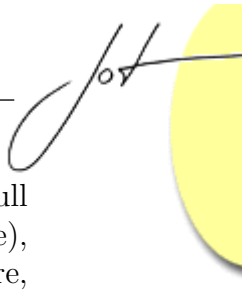
Dynamically Loaded Attributes

If an attribute of a particular type is loaded dynamically (e. g., `val` defined in module `cache`, cf. Sec. 9), all object descriptors belonging to this type must be extended by a new slot corresponding to this attribute. This can be achieved by iterating through the descriptor repository and reallocating each descriptor. To make sure that the descriptor pointers of objects remain stable in that case, an additional level of indirection is used that is not shown in Fig. 9. Since loading new attributes is not expected to happen very frequently, the effort for these reallocations is acceptable. In order to reduce it even further, descriptors are always allocated with a certain number of extra slots in advance (indicated by the dashed boxes in Fig. 9).

Empty and Null Objects, Object Deletion

Initially, the descriptor repository of each open type contains a single entry at position zero called the *empty descriptor*, which does not possess any attributes, i. e., all its slots initially contain dashes.

An empty object is represented by an intermediate cell referencing this empty descriptor and a unique empty data area (which will never be accessed). Similarly, a null object is represented by an intermediate cell referencing the empty descriptor,



too, but with a null data pointer. Thus, attribute inspections on empty and null objects always return the same result (i. e., null values of the attribute's target type), without requiring any additional run time checks to detect null objects. Furthermore, an object can be easily and efficiently deleted (even while it is still referenced) by simply deallocating its data area, setting its data pointer to null, and replacing its descriptor pointer with a pointer to the empty descriptor. Finally, since every empty object possesses a unique empty data area, comparisons for object equality can simply compare the object's data pointers for identity.

Precompiler for C++

The definitions of open types, attributes, and relationships are transformed into suitable C++ type and function definitions by a rather simple precompiler based on the EDG C++ Front End (cf. `www.edg.com`). For example, an open type is transformed to a C++ class, while an attribute is transformed to two or three global virtual functions, as described in Sec. 7.3 (and GVFs are in turn transformed to regular functions plus some auxiliary data structures, as described in [14, 16]). Usage of the attribute inspection operator `@` in an expression is transformed to a call of the corresponding get function, while mutator (and attribute-initialization constructor) calls are transformed to calls of the corresponding update function. The implementation of anonymous and automatic attributes, in particular the way to tell the underlying C++ compiler to automatically apply automatic attributes transitively, requires some sophisticated tricks whose description is beyond the scope of this paper.

Automatic Garbage Collection

To perform automatic garbage collection (GC) for open type objects, all global and local variables of open types are treated as root pointers, whose addresses are added to resp. removed from a global root table by the constructors resp. destructors of the class representing the type.[14] Furthermore, the values of all attributes whose target type is an open type, too, are treated as inter-object pointers, whose location in an object's data area can be determined from the object's descriptor.

Based on this information, any tracing GC algorithm can be used to reclaim the storage of objects which have become unreachable. By overloading the assignment operator of open types to perform appropriate extra actions in addition to copying the pointer to the intermediate cell, it is even possible to implement an incremental GC algorithm. In particular, an incremental Mark&Sweep algorithm based on [8] has been implemented.

[14] If open type values are dynamically created with the C++ `new` operator, these are treated as root pointers, too.

Performance Results

To compare the run time efficiency of the open type implementation (including its garbage collector) with standard class-based object-oriented systems, a simple test program creating, inspecting, and manipulating randomly structured object graphs has been written in standard C++, Java, and C+++ (i. e., C++ with open types). In C++, objects which are no longer needed, are explicitly deleted, while in C+++ and Java automatic garbage collection is employed for that purpose. By changing the values of some configuration parameters, the relative frequencies of object creations, inspections, and manipulations can be modified to obtain quite different application characteristics with a single program.

The main results of comparing the overall run times of these programs with different parameter settings can be summarized as follows:

- C+++ is between 1.9 and 2.5 times slower than C++.
  Since C/C++ is usually regarded as a performance yardstick, it is not really surprising that the current precompiler-based C+++ implementation cannot really compete with it. On the other hand, a slowdown of this magnitude might still be acceptable – in exchange for the significantly improved flexibility provided by open types – for many applications where performance is not really critical.

- C+++ is up to 2.5 times faster than Java running on the Hotspot Virtual Machine.
  When considering the fact that over many years lots of people have spent considerable effort to improve the performance of Java implementations in general and garbage collection techniques in particular, this is both a surprising and a quite encouraging result.

Of course, to obtain more substantial and detailed performance data and to further improve the C+++ implementation, more fine-grained measurements with different benchmark and real-world programs are required. For now, however, it has been sufficient to prove that open types (as well as global virtual functions and types with null values) can be implemented with quite acceptable run time performance.

11 RELATED WORK

Since open types and bidirectional relationships constitute the genuine contribution of this paper, while the accompanying issues of null values, global virtual functions, and modules have already been published and discussed elsewhere [15, 14, 16], the following discussion of related work is confined to the former.

Extensible Data Structures

As already mentioned in Sec. 1, *aspect-oriented programming languages* such as AspectJ [22] and AspectC++ [32] provide so-called inter-type member declarations or introductions to retroactively extend existing data structure definitions without needing to change the code of the original definitions. Nevertheless, however, some kind of recompilation or "weaving" is required by all these approaches: While the AspectC++ compiler needs the source code of the original definition together with all extension code to produce a new definition that is actually compiled by a C++ compiler, the AspectJ compiler is able to perform the combination on the byte code level. Frameworks such as JMangler [23] are even able to delay the final composition until load time, but in either case a class remains fixed once it has been loaded. In particular, all objects of a class possess the same storage structure and size, and it is impossible to change this at run time. Open types, on the other hand, allow attributes to be loaded dynamically, even for types which have already been instantiated. Actually, aspect-oriented approaches still adhere to the traditional concept of records as fixed data structures and only make their definition more flexible, while open types support truly flexible objects whose storage size might vary over time. Furthermore, as already mentioned in Sec. 8.4, extensions provided by different aspects might conflict with each other, while attributes defined in different modules will always be compatible.

Subject-oriented programming [13] and *multi-dimensional separation of concerns* [34] also allow data structures to be specified and composed very flexibly, supporting even the merge of independently developed classes into a single new one. Again, however, all resulting classes remain fixed while a program is being executed.

The *Common Lisp Object System* (CLOS) [21] and languages such as Dylan [7] based on similar principles deviate from the approach of other object-oriented languages that everything belonging to a class must be defined (or at least declared) in the class, by associating methods with *generic functions* and allowing them to be defined separately and incrementally. However, the set of data fields making up a class must still be defined at once and cannot be extended later, except by redefining the whole class. In contrast, open types apply the "generic function principle," i. e., the possibility to define methods separately and independently, to data fields, too.

MultiJava [6, 27] is a recent extension of Java that also addresses the problem of retroactively extending existing classes in a strictly modular way, but again only augmenting methods is supported, while the data fields making up a class remain fixed.

Dynamically typed languages and systems with integrated interactive programming environments such as Smalltalk [11] allow the user to extend an existing class by adding new instance variables and automatically reallocate all currently existing instances to contain space for the new variables. While the effort for these reallocations might be acceptable if such extensions are not performed very frequently, the open type implementation avoids it by reallocating only the descriptors of an

open type instead of all its objects when a new attribute is loaded dynamically. Afterwards, individual objects are only extended on demand, i. e., when they actually receive a value for the new attribute.

Prototype-based languages such as Self [36] offer many of the advantages of open types over record-oriented data models, e. g., the possibility to incrementally define attributes (called *slots*) and inheritance relationships (via *parent slots*) as well as support for dynamic object evolution (by dynamically adding or removing slots). Even the implementation of objects in Self based on *maps* [4] is similar to that of open type objects based on descriptors. However, since object reallocations are not expected to occur frequently, objects are addressed directly instead of through an intermediate cell, requiring a scan through the entire heap to update all references to an object if it has to be actually moved. Furthermore, no attempt is made to find and possibly reuse an existing map if an object is extended by a new slot. Since a new object is usually created by cloning an existing prototype, and afterwards it is only extended in rare cases to set up a new "clone family," these drawbacks are acceptable. Open type objects, however, are always created as empty objects which are frequently extended, at least in the course of their initialization. Therefore, it is vital to perform this operation rather efficiently.

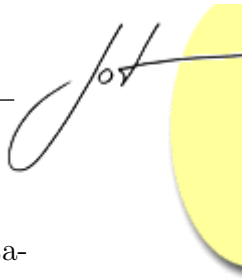
Apart from these implementation details, the key difference between Self and open type objects is the fact that the latter are statically type-safe, i. e., they combine the flexibility of dynamically typed systems with the safety of statically typed languages.

Bidirectional Relationships

Even though bidirectional relationships between data types are omnipresent in various kinds of applications, and modeling languages such as the Entity-Relationship model [5] and UML [29] support them as an integral part, the possibility to directly define them in a programming language, whose run time system keeps their directions consistent automatically, is hard to find.

RelJ [2] is a prototypical language consisting of a "middleweight" fragment of Java plus support for relationships as first-class citizens. Similar to open types, relationship definitions are separated from class definitions and therefore can be added incrementally. The way to establish and remove relationships between objects is quite similar to mutator calls for open type objects, and the rules to enforce the restricted multiplicities of 1:1, 1:N, and N:1 relationships are pretty much the same.

In addition to a source and a target type, a relationship might possess its own data fields and methods, it might be part of a separate inheritance hierarchy, and it might itself take part in another relationship. Since data fields, methods, and inheritance are not directly provided for open types, however, it would be logically inconsistent to provide them for relationships between them. Instead, one might allow attributes and relationships as source or target types of other attributes and relationships and possibly as parameter types of functions in order to provide com-



parable features for open types.

While RelJ's main contribution so far is a sound formal model, no implementation ideas have been published yet. In fact, to provide an efficient implementation of the model, a flexible data representation format similar to that of open types appears to be necessary to allow information about an object's relationships to be stored directly within the object.

Description logic systems such as Classic [3] and Loom [25] provide data models which are very similar in nature to open types and have in fact influenced some of their ideas. They provide *concepts* (corresponding to open types) and *roles* (corresponding to attributes and relationships), which are defined separately and independently, and roles might possess *inverse roles*. Furthermore, a running system can be extended by new definitions at any time.

However, since description logic systems are actually AI tools, providing powerful reasoning capabilities such as subsumption checking, automatic instance classification, and truth maintenance, using them as bare data models of a programming language would mean to break a fly upon the wheel. Therefore, open types might be viewed as the result of reducing a description logic system to a simple data representation system by stripping off all AI functionality.

Inheritance and Aggregation

Supporting *multiple and repeated inheritance* in a way that combines high expressiveness and flexibility with comprehensibility, ease of use, and efficient implementability has always been a great challenge for object-oriented language designers. In particular, the question whether and under which circumstances a multiply inherited entity (i. e., a data field, a method, or a whole class) shall be replicated in a derived class or not, is hard to answer in general. Furthermore, if entities are replicated, the resulting name clashes have to be resolved in some satisfactory way.

The proposed solutions range from totally rejecting multiple inheritance (e. g., in Smalltalk [11]) over restricting it to inheritance of interfaces (e. g., in Java [1]) to supporting it in a rather general way (e. g., in Eiffel [26], C++ [33], and Timor [19, 20]) with quite different syntactical and semantical forms. In any case, however, numerous specialized language constructs and partially sophisticated rules are required to support at least the most frequently occurring cases in practice. With open types, however, the already present and comparatively simple concept of bidirectional relationships can be used to model virtually all kinds of multiple and repeated inheritance in a straightforward manner, without requiring any additional language constructs and rules for that purpose.

Again, the approach taken in Self [36] is quite similar to that of open types: Instead of introducing special language constructs and rules, the already present concept of slots can be employed to distinguish, e. g., between replicated and shared inherited entities.

In addition to significantly simplifying inheritance issues, the concept of automatic attributes and relationships unifies the concepts of *aggregation* (expressed by normal attributes and relationships), *inheritance* (expressed by automatic attributes and relationships), and user-defined *type conversions* (expressible by automatic attributes whose get function is redefined to perform the appropriate conversion). Even though related, these concepts are usually clearly separated from each other in other programming languages. C++, for example, provides two different ways to specify user-defined type conversions, i. e., constructors and conversion operators [33], in addition to implicit type conversions based on subtype polymorphism (where the latter are applied transitively if necessary to convert a derived class to an indirect base class, while the former are not). The combination of these different mechanisms leads to rather complicated rules about the applicability of conversions and the ranking of conversion sequences that is needed for overload resolution. In contrast, automatic attributes constitute a single coherent concept which subsumes and unifies these different ones.

Other approaches that aim at reconciling the partially conflicting concepts of aggregation and inheritance are *parts inheritance* in Timor [19] and *compound references* in [30].

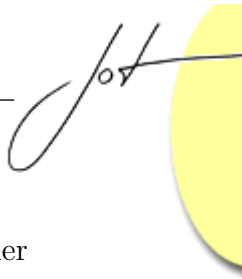
The fact that an attribute is actually a set of functions with a predefined implementation that can be changed by the programmer, corresponds (amongst others) to the concept of *abstract variables* also found in Timor [18] and again to the concept of slots in Self. However, open types go a step further by allowing these functions to be redefined multiple times, where each new implementation can call the previous one on demand.

Variable functions proposed by Odersky [28] are quite similar to attributes of open types by constituting an updatable mapping from a source to a target type that might be accessed either in an object-oriented (`x.y` resp. `x@y`) or in a functional notation (`y(x)`). The cited paper also contains some general remarks about possible implementations, but no ideas comparable to the sophisticated attribute management of open types have been found.

12 CONCLUSION

Summary

Open types have been presented as a simple yet powerful data model for statically typed procedural and object-oriented programming languages, that overcomes the limitations of the traditional record-oriented model, especially the problem of retroactively extending existing data types with new attributes. Even though the core concepts of open types and attributes are hardly more complex than simple record types, their extension with bidirectional relationships and automatic and anonymous attributes and relationships leads to a remarkable degree of expressiveness and flexibility. In particular, virtually all kinds of object-oriented inheritance



can be modeled in a rather straightforward way.

By complementing open types with global virtual functions and modules, other core concerns of object-orientation, i. e., dynamic binding and encapsulation, are covered, too. In summary, the combination of these concepts leads to *advanced procedural programming languages*, which are at the same time conceptually simpler and more expressive than their object-oriented counterparts. In particular, some important features provided by aspect-oriented languages, such as inter-type member declarations and advice, are available for free, i. e., without requiring additional language constructs.

Current Status and Future Work

The basic ideas of open types and virtual functions have been developed, implemented, and refined over several years. Recently, they have been successfully employed in a small student project (2500 lines of C+++ code), a larger master's project (4000 lines of C+++ code), and in re-implementing the C+++ precompiler in C+++ itself (4000 lines of C+++ code providing truly modular extensions to several hundred thousand lines of existing C code).

In particular, the re-implementation of the precompiler based on the EDG C++ Front End (cf. Sec. 10.4) has proven that the basic idea of non-invasively extending and modifying an existing software system is actually practicable, even if the original system has not been specifically designed with that aim. Furthermore, it has demonstrated that it is easily possible to automatically turn ordinary statically bound C functions into dynamically bound global virtual functions which can be redefined on demand, i. e., to actually integrate "legacy" code into C+++ programs.

Even though the current implementation of C+++ already exhibits quite acceptable performance, the potential for optimizations is surely not exhausted yet. By fine-tuning critical parts of the attribute management system and the garbage collection algorithm, additional performance gains are expected.

REFERENCES

[1] K. Arnold, J. Gosling, D. Holmes: *The Java Programming Language* (Third Edition). Addison-Wesley, Boston, 2000.

[2] G. Biermann, A. Wren: "First-class Relationships in an Object-oriented Language." In: A. P. Black (ed.): *ECOOP 2005 – Object-Oriented Programming* (19th European Conference; Glasgow, UK, July 2005; Proceedings). Lecture Notes in Computer Science 3586, Springer-Verlag, Berlin, 2005, 262–286.

[3] R. J. Brachman, D. L. McGuinness, P. F. Patel-Schneider, L. A. Resnick: "Living with CLASSIC: When and How to Use a KL-ONE-Like Language." In:

J. F. Sowa (ed.): *Principles of Semantic Networks. Explorations in the Representation of Knowledge*. Morgan Kaufmann Publishers, San Mateo, CA, 1991, 401–456.

[4] C. Chambers, D. Ungar, E. Lee: "An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes." In: N. K. Meyrowitz (ed.): *4th Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (New Orleans, LA, October 1989). *ACM SIGPLAN Notices* 24 (10) October 1989, 49–70.

[5] P. P. Chen: "The Entity-Relationship Model – Toward a Unified View of Data." *ACM Transactions on Database Systems* 1 (1) March 1976, 9–36.

[6] C. Clifton, G. T. Leavens, C. Chambers, T. Millstein: "MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java." In: *Proc. 2000 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '00)* (Minneapolis, MN, October 2000). *ACM SIGPLAN Notices* 35 (10) October 2000, 130–145.

[7] I. D. Craig: *Programming in Dylan*. Springer-Verlag, London, 1997.

[8] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, E. F. M. Steffens: "On-the-Fly Garbage Collection: An Exercise in Cooperation." *Communications of the ACM* 21 (11) November 1978, 966–975.

[9] M. Ernst, C. Kaplan, C. Chambers: "Predicate Dispatching: A Unified Theory of Dispatch." In: E. Jul (ed.): *ECOOP'98 – Object-Oriented Programming* (12th European Conference; Brussels, Belgium, July 1998; Proceedings). Lecture Notes in Computer Science 1445, Springer-Verlag, Berlin, 1998, 186–211.

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[11] A. Goldberg, D. Robson: *Smalltalk-80. The Language*. Addison-Wesley, Reading, MA, 1989.

[12] D. Goldberg: "What Every Computer Scientist Should Know About Floating-Point Arithmetic." *ACM Computing Surveys* 23 (1) March 1991, 5–48.

[13] W. Harrison, H. Ossher: "Subject-Oriented Programming (A Critique of Pure Objects)." In: *1993 Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (September 1993), 411–428.

[14] C. Heinlein: "Virtual Namespace Functions: An Alternative to Virtual Member Functions in C++ and Advice in AspectC++." In: *Proc. 2005 ACM Symposium on Applied Computing (SAC)* (Santa Fe, New Mexico, March 2005), 1274–1281.

[15] C. Heinlein: "Null Values in Programming Languages." In: H. R. Arabnia (ed.): *Proc. Int. Conf. on Programming Languages and Compilers (PLC'05)* (Las Vegas, NV, June 2005), 123–129.

[16] C. Heinlein: "Global and Local Virtual Functions in C++." *Journal of Object Technology* 4 (10) December 2005, 71–93.

[17] *JBoss AOP*. http://www.jboss.org/products/aop.

[18] J. L. Keedy, G. Menger, C. Heinlein: "Taking Information Hiding Seriously in an Object Oriented Context." In: *Net.ObjectDays 2003. Tagungsband* (Erfurt, Germany, September 2003). tranSIT GmbH, Ilmenau, 2003, ISBN 3-9808628-2-8, 51–65.

[19] J. L. Keedy, C. Heinlein, G. Menger: "Inheriting Multiple and Repeated Parts in Timor." *Journal of Object Technology* 3 (10) November/December 2004, 99–120.

[20] J. L. Keedy, C. Heinlein, G. Menger, M. Evered: "Diamond Inheritance and Attribute Types in Timor." *Journal of Object Technology* 3 (10) November/December 2004, 121–142.

[21] S. E. Keene: *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, Reading, MA, 1989.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold: "An Overview of AspectJ." In: J. Lindskov Knudsen (ed.): *ECOOP 2001 – Object-Oriented Programming* (15th European Conference; Budapest, Hungary, June 2001; Proceedings). Lecture Notes in Computer Science 2072, Springer-Verlag, Berlin, 2001, 327–353.

[23] G. Kniesel, P. Costanza, M. Austermann: "JMangler – A Powerful Back-End for Aspect-Oriented Programming." In: R. E. Filman, T. Elrad, S. Clarke, M. Aksit (eds.): *Aspect-Oriented Software Development*. Pearson International, 2004, 311–342.

[24] A. Koenig, B. Stroustrup: "Foundations for Native C++ Styles." *Software—Practice and Experience* 25 (S4) December 1995, 45–86.

[25] R. MacGregor: "The Evolving Technology of Classification-Based Knowledge Representation Systems." In: J. F. Sowa (ed.): *Principles of Semantic Networks. Explorations in the Representation of Knowledge*. Morgan Kaufmann Publishers, San Mateo, CA, 1991, 385–400.

[26] B. Meyer: *Eiffel: The Language*. Prentice-Hall, New York, 1994.

[27] T. Millstein, M. Reay, C. Chambers: "Relaxed MultiJava: Balancing Extensibility and Modular Typechecking." In: *Proc. 2003 ACM SIGPLAN Conf. on*

Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '03) (Anaheim, CA, October 2003). *ACM SIGPLAN Notices* 38 (11) November 2003.

[28] M. Odersky: "Programming with Variable Functions." In: *Proc. Int. Conf. on Functional Programming* (Baltimore, Sept. 1998).

[29] OMG: *OMG Unified Modeling Language Specification* (Version 1.5). Object Management Group, March 2003.

[30] K. Ostermann, M. Mezini: "Object-Oriented Composition Untangled." In: *Proc. 2001 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '01)* (Tampa Bay, FL, October 2001). *ACM SIGPLAN Notices* 36 (11) November 2001, 283–299.

[31] D. L. Parnas: "On the Criteria to Be Used in Decomposing Systems into Modules." *Communications of the ACM* 15 (12) December 1972, 1053–1058.

[32] O. Spinczyk, A. Gal, W. Schröder-Preikschat: "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language." In: J. Noble, J. Potter (eds.): *Proc. 40th Int. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS Pacific)* (Sydney, Australia, February 2002), 53–60.

[33] B. Stroustrup: *The C++ Programming Language* (Special Edition). Addison-Wesley, Reading, MA, 2000.

[34] P. Tarr, H. Ossher, W. Harrison, S. M. Sutton Jr.: "N Degrees of Separation: Multi-Dimensional Separation of Concerns." In: *Proc. 21st Int. Conf. on Software Engineering* (May 1999), 107–119.

[35] M. Torgersen: "The Expression Problem Revisited. Four New Solutions Using Generics." In: M. Odersky (ed.): *ECOOP 2004 – Object-Oriented Programming* (18th European Conference; Oslo, Norway, June 2004; Proceedings). Lecture Notes in Computer Science 3086, Springer-Verlag, Berlin, 2004, 123–143.

[36] D. Ungar, R. B. Smith: "Self: The Power of Simplicity." In: *2nd Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (Orlando, FL, October 1987). *ACM SIGPLAN Notices* 22 (12) December 1987, 227–241.

[37] N. Wirth: "The Programming Language Oberon." *Software—Practice and Experience* 18 (7) July 1988, 671–690.

ABOUT THE AUTHORS

Christian Heinlein has been working as a Scientific Assistant at the University of Ulm, Germany, where he conducted the research project APPLEs, that aims at developing "Advanced Procedural Programming Languages," which are both conceptually simpler and more flexible than standard object-oriented languages. He can be reached at christian.heinlein@uni-ulm.de. See also www.informatik.uni-ulm.de/rs/mitarbeiter/ch/apples.