Values in Object Systems

Dirk Bäumer, Dirk Riehle, Wolf Siberski,
Carola Lilienthal, Daniel Megert, Karl-Heinz Sylla, Heinz Züllighoven

Ubilab

Ubilab Technical Report 98.10.1

Ubilab

Ubilab is the information technology laboratory of UBS AG. It pursues a small number of attractive, highly competitive research projects with the aim of maintaining the status of a recognized research institution.

It is the task of Ubilab to actively assist UBS with its goal of becoming a leader in the mastery of modern IT equipment, techniques, and methods. This task is pursued by means of intimate involvement in application-oriented research and advanced development projects. Furthermore, UBS contributes to fostering the interaction between IT research and application, that is between academia in its search for new methods, and business in its application of them. To this purpose Ubilab cooperates with universities and other research institutions on a worldwide basis. The goal is to expand the scope of the Laboratory by carrying out common projects, thus guaranteeing—provided top-notch partners are found—the quality of the research. The impact of Ubilab is therefore focussed on both the IT departments of UBS AG and the IT research community at large.

For further information about Ubilab, staff, projects, or publications see our World Wide Web (WWW) pages. Feel free to contact the staff personally via e-mail or to write to the mailing address given below.

Location of Ubilab

Universitätsstrasse 84
CH-8033 Zurich
Switzerland

Mailing address

UBS AG, Ubilab
Postfach
CH-8098 Zurich

Electronic access

Phone: ++41 1 236 57 14
Fax: ++41 1 236 46 71
E-mail: firstname.lastname@ubs.com
WWW: http://www.ubs.com/ubilab

Table of Contents

1 Introduction 7

2 Motivation 9

2.1 MonetaryAmount example 9

2.2 Performance consequences 10

3 Objects and Values 13

3.1 Customer and Account example 13

3.2 Values and value types 14

3.3 Objects and object types 15

4 Objects and Values in Modeling 17

4.1 Implementation techniques 17

4.2 The Immutable Object approach 18

4.2.1 The Copy-On-Write Approach 19
4.2.2 Client-Side copy 19
4.2.3 The Body/Handle idiom 20

4.3 Programming languages 21

5 Project Experiences 23

5.1 GeBOS 23

5.2 Geo 24

5.3 KMU Desktop 24

5.4 Summary 24

6 Related work 25

6.1 User defined data types in C++ 25

6.2 Smalltalk 26

6.3 Java 26

6.4 The proposed CORBA standard for “Objects By Value” 27

6.5 Other distributed systems 27

7 Conclusions 29

References 31

Abstract

Objects and values are fundamental yet complementary concepts of software system modeling and implementation. However, in the context of large object systems, modeling and implementing value types has received no sufficient attention. Our experiences show that proper understanding of value types can ease programming and improve performance of such systems significantly. In this paper, we discuss the impact of value types on object-oriented system design, implementation, and performance. We discuss several implementation techniques that let us introduce value types into systems implemented in mainstream object-oriented programming languages.

Dirk Bäumer, Takefive Software AG. Eidmattstrasse 51, 8032 Zurich, Switzerland.
Email: baeumer@takefive.ch

Dirk Riehle, UBS AG, Ubilab. Postfach. CH-8098 Zurich.
E-mail: Dirk.Riehle@ubs.com or riehle@acm.org

Wolf Siberski, RWG GmbH. Räpplenstraße 17, 70191 Stuttgart, Germany.
E-mail: Wolf.Siberski@rwg.de

Carola Lilienthal, University of Hamburg. Vogt-Kölln-Str. 30, 22527 Hamburg, Germany.
Email: Carola.Lilienthal@informatik.uni-hamburg.de

Daniel Megert, UBS AG, Ubilab. Postfach. CH-8098 Zurich.
E-mail: Daniel.Megert@ubs.com or Daniel.Megert@acm.org

Karl-Heinz Sylla, GMD. Schloß Birlinghoven, 53754 Sankt Augustin, Germany.
E-mail: sylla@gmd.de

Heinz Züllighoven, University of Hamburg. Vogt-Kölln-Str 30, 22527 Hamburg, Germany.
Email: Heinz.Zuellighoven@informatik.uni-hamburg.de

Chapter 1

Introduction

Objects and values are complementary modeling and implementation concepts, used in every modern object-oriented software system. Frequently, the concept of value in the context of object-oriented systems is reduced to mean “primitive data types” like integer, float, or string. Yet, the wealth of research and practice on applicative systems demonstrates that values are powerful modeling and implementation concepts, which offer many advantages over objects.

The most prominent advantage of values over objects is that values are side-effect free. Since values cannot be referenced, they cannot be shared in different contexts, omitting many of the side-effect problems that haunt large object systems. From this primary property of values, many secondary properties can be derived that ease their handling in such diverse domains as serialization, persistence, querying, distribution, and concurrency. These properties, illustrated in section 2, can significantly ease system design and implementation as well as increase system performance.

So-called “pure” object-oriented programming languages and systems suggest to model and implement everything as an object. Our experiences in a number of large-scale projects suggest that this approach is counterproductive, if not outright harmful. In our projects, we therefore do not only work with object types, but also with domain-specific value types to overcome the problems a naive object model causes.

In this paper, we analyze the properties of values in the context of large object systems. We motivate how value types help make a system more understandable and how they help increase performance. Finally, we show how value types can be implemented in a number of different ways, targeting different mainstream programming languages and systems.

Section 2 provides a number of motivating examples for using values in large object systems. Section 3 describes the properties of value types in object systems and compares them with object type properties. Section 4 analyzes the design and implementation space of value types in object systems and provides several implementation techniques. Section 5 reports on our experiences with using value types in object systems in a number of different large projects, mainly from the banking domain. Section 6 presents related work and analyzes how values have been handled so far in mainstream programming languages and systems. Section 7 concludes the paper.

Chapter 2

Motivation

We now motivate the use of values in object-oriented systems by showing the problems a naive object-oriented approach typically causes. This encompasses both the general problem of side-effects, as well as design, implementation, and performance problems in such diverse domains as distribution, serialization, and persistence.

2.1 MonetaryAmount example

In the banking domain, MonetaryAmount is a fundamental abstraction. It provides a currency and an amount. The currency is typically represented as a three-letter string like “USD”, “DEM”, or “CHF”. The amount is typically represented as a double floating point number. An example use of MonetaryAmount is an Account class, which models its balance using instances of the MonetaryAmount abstraction.

The standard approach is to model MonetaryAmount as a class, so that a concrete monetary amount like “5.00 CHF” is represented as an object. Assume now, that MonetaryAmount is a class that provides operations, which let clients alter the state of a monetary amount object. An account object, when asked for its balance, returns a monetary amount object. If the object references hold by the client and the account object point to the same monetary amount object, the client might alter the account’s balance by changing the amount object, thereby encompassing all restrictions the account object might have imposed on changing its balance. This is the problem of side-effects through aliasing.

To overcome this seemingly technical problem, a number of solutions can be applied. For example, account objects might always return copies of their balance object. Or, a Body/Handle idiom based implementation of amount objects might copy their body upon first write access. Or, the amount object’s interface might only provide operations that prohibit altering its state.

All these specific solutions point to a common theme: MonetaryAmount is best modeled as a value type rather than as an object class. The next section discusses properties of value types and value semantics in object-oriented systems. Here, it suffices to emphasize that values do not suffer from side-effects, because conceptually speaking, they have no alterable state. Typically, they are implemented

as immutable objects, but this need not always be the case (see the section on implementation techniques for value types).

2.2 Performance consequences

Modeling abstractions like MonetaryAmount as side-effect free value types has a number of advantages compared to modeling them as object types.

In *distributed systems*, object references might cross process boundaries. If not taken care of properly, an object constellation might occur in which an account object references its monetary amount object in a different process. This might cause frequent cross-process calls, significantly reducing performance. While obviously a naive approach, many products on the market and many publications suggest a fully transparent approach to distribution. If monetary amounts are modeled and implemented as values rather than objects, they can always be copied with their enclosing objects, thereby preventing performance degradation.

In distributed systems, a number of different approaches are known to overcome these problems. For example, parameters may be passed using call-by-visit, or objects may be made immutable, or systems might provide elaborate object migration schemes where not only single objects but full object graphs are migrated in to order reduce the number of cross-process calls. Making value types explicit in distributed systems provides a conceptually clean argument for many if not all aspects of these approaches.

When using *multithreading*, modeling with value types either reduces or fully avoids locking overhead (and related resource consumption problems). Conceptually, values have no alterable state, so they can be implemented as immutable objects, avoiding the need for locking. Other implementation techniques (see section 4), at least partially reduce the problem: if a value is represented as an object, copy semantics can be assumed, avoiding locking overhead. If a value type is implemented using the Body/Handle idiom, the locking overhead can be avoided at least for the client.

When *serializing objects*, i.e., turning them into passive data representations, the serialization algorithm must cater for cyclic references between objects. Every check, whether an object reference introduces a cycle, slows down the execution of the serialization algorithm. If it is clear than an entity, represented as an object, is conceptually a value, it need not be taken care of as a possible cyclic reference. Rather, the value can be directly written to the buffer. For every value, the check for cyclic references is omitted. Because the number of “value objects” in a system can be very high (see experience section), typically much higher than the number of regular objects, performance can increase significantly.

More importantly, there is no need to maintain object ids for values. This makes the data buffer footprint of the value smaller than the one of a comparable object. The value can be dumped directly from main memory, without further analysis of its structure and without having to check for embedded object references.

These advantages apply primarily to lightweight values, i.e., values with a small implementation state. Heavyweight values compromise these advantages, because they might be implemented based on object references that need to be followed. In addition, serializing complex object graphs with many equal values might have an unexpectedly large buffer footprint, if the values are heavyweight and copied rather than referenced within the buffer.

When *making objects persistent*, be it in a relational or an object database, objects incur additional management overhead. In relational databases, every object, as small as it may be, typically requires putting it into a table of its own. Subsequent activation causes costly joins, degrading performance

once more. If it is clear that an object is conceptually a value, it can be embedded efficiently within its enclosing object. Then, the value can both be read and written together with its enclosing object. The object, and all its value attributes, may be maintained as a single tuple in one table rather than spread over several tables.

Many database systems today offer functionality that points in this direction. For example, some databases like Sybase or Oracle extend the set of generally accepted value types (integer, string, etc.) with further value types like date and time, which are typically not considered values but rather objects. These new value types are directly embedded as part of a traditional database entry or an object mapped on such an entry. Being explicit that these are value types, and letting users introduce their own value types, can significantly boost performance for the reasons just given.

Again, these arguments primarily apply to lightweight values. Heavyweight value types might require their own table, much like objects do. It becomes more difficult, if values are to be treated polymorphically. Then, they either need to be referenced, causing possibly costly joins like regular objects, or they are stored as a binary large object block (blob), which prevents querying them.

When *querying relational databases*, value types are also of help. They simplify queries and make efficient index generation possible. If, for example, the monetary amount of a class `Account` is modeled as an object, an SQL query like `select from Account where MonetaryAmount = '100 SFR'` has to be realized as a join of the tables `Account` and `MonetaryAmount`. Furthermore it is impossible to use the standard indexing mechanism of a relational database engine to create a `MonetaryAmount` index for the table `Account`. This causes further performance degradation.

These examples illustrate that modeling with value types where adequate can significantly improve the performance of object-oriented software systems in a wide area of technical concerns.

Chapter 3

Objects and Values

We now review the properties of objects and values as complementary modeling concepts. These two concepts are at the heart of every modeling language, domain model, and system implementation.

3.1 Customer and Account example

To better illustrate the use of values in the context of objects, we extend our initial example. In the banking domain, we model customer information as an instance of class Customer, which is an object type. Customer has two attributes, one called name, which is of value type PersonName, and (for simplicity's sake) one attribute of value type reference to object of class Account (object references are values). Class Account, which is an object type, also has two attributes, one called balance, which is of value type MonetaryAmount, and one called accountNo, which is of value type AccountNumber. Figure 1 illustrates the example design.

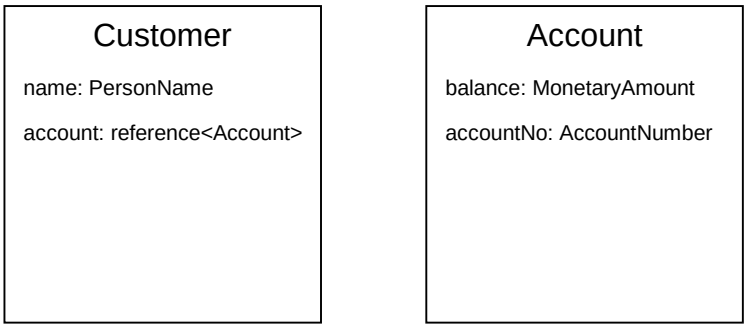

Figure 1: Design example

3.2 Values and value types

Values have four key properties V1-V4. We have taken these properties from MacLennan's seminal article on objects and values in programming languages [Mac82] and adapted them to the modeling level.

- Values are abstractions (universals or concepts) which model abstractions from a problem domain.
- Values have no lifecycle (i.e., they do not exist in time, are not created nor changed nor deleted).
- Values have no alterable state; representations can only be interpreted, not changed.
- Values are referentially transparent (i.e., there are no side-effects of using a value on other parts of the system).

Examples of values are integers, real numbers, strings, dates, social security numbers, currencies, monetary amounts, account numbers, etc. Values in computing systems are instances of value types, for example Integer, Float, String, Date, SocialSecurityNumber, Currency, MonetaryAmount, and AccountNumber.

Properties V1 (modeling abstraction from some domain) and V2 (no lifecycle) apply to value types on every level. Values live in an inaccessible "universe of values." A key consequence is that values are accessible to us only through representations, which may take on arbitrary forms. In fact, we only learn about values through their representations.

Property V3 needs further explanation and some refinement. A value becomes accessible only through its representations. For every value type, there can be any number of different representation schemes, and for every value, there can be any number of different representations. For example, different representations of the value two are "2" and "two". Typically, different representations serve different purposes. Given knowledge of the respective representation schemes, both representations can be interpreted as pointing towards the same intangible value two. The association between a representation and its corresponding value cannot be changed. (We use "a representation" to denote both an occurrence of a representation and the representation itself [WJ95]).

Property V4 eventually distinguishes values from objects. There are no references to values, because there are only bindings of (conceptually) immutable representations.

In programming languages, value types are offered as so-called built-in types or (primitive) data types. The most common examples are integers, floating point numbers, characters, etc. These value types made it into programming languages very early, because they are the domain-specific value types of engineering and mathematics (FORTRAN) and transaction-oriented business processing (COBOL). We call them *primitive value types,* not so much because they are primitive in any respect, but because their standard representation schemes provide good constructs to build representation schemes for other value types.

Given no specific programming language concept, value types have to be implemented as classes, for example as classes Date, SocialSecurityNumber, or MonetaryAmount. From a conceptual point of view, we are still dealing with value types, be they implemented as classes or not. We call an instance of a value type a "value object". Cunningham demonstrates the importance of treating value objects properly: the information integrity of applications closely depends on handling value objects with value semantics [Cun95]. We call value types which require implementation constructs like classes *non-primitive value types.*

A particularly important value type is object identity. Every object has a unique identity, through the representations of which it may be referenced. A reference is a representation of an object identity using a specific representation scheme. Representation schemes may vary drastically, depending on

their intended use (e.g. main memory addresses, remote references with location encoding, or globally unique object identifiers).

The compiler-generated code (or the runtime system, or the virtual machine) must be able to directly interpret at least one representation scheme for object identity (typically a memory address) in order to invoke operations on the referenced object. Here, the need for providing ways to introduce new representations of new value types or extending old ones becomes particularly obvious: Every distributed system and every object-oriented database introduces its own proprietary scheme for changing and extending the semantics of the de-referencing process of object references. A coherent approach, defined on the programming language level, would help to do so in an efficient manner and avoid integration headaches like those of distribution with databases.

Values are frequently implemented with “copy semantics”, which means that they are always copied so that no referential integrity has to be maintained. However, this is a particular implementation technique, and other techniques could be used equally well (see section on implementing value types).

3.3 Objects and object types

Objects exhibit the following properties O1-O4 (again adopted and adapted from [Mac82]):

- Objects are representations of phenomena from a problem domain.
- Objects have a life cycle (i.e., they exist in time, can be instantiated, changed, and deleted).
- Objects have identity that unambiguously denotes them (thus, they can be referenced).
- Objects can be shared, which is a consequence of that objects can be referenced.

Examples of objects are ubiquitous. Objects are instances of classes. A class is the abstraction from several similar phenomena; it defines what can be done with its instances. We do not discuss object properties in general, but only compare them with value types.

The phenomenon an object represents may be tangible or intangible. Problem domains may both be technical or business domains (property V1 and O1). Both objects and values (and classes and value types) are modeling concepts (i.e., they are not restricted to implementation).

While values exist independently of time in an invisible universe of values, objects do exist in time, independently of whether these objects are representations of tangible or intangible phenomena (properties V2 and O2). As a consequence, one needs to manage the life cycle of objects, defining what it means for an object to come into existence, change over time, and be eventually destroyed. The importance of life-cycle management is exemplified by the existence of standards (e.g. the CORBA life-cycle service [OMG95]). No such complications exist for values.

Because objects exist in time, they can be distinguished from each other by their identity (property O3). A value, in contrast, has no identity, because there are only occurrences of representations, which are never identical (property V3). Representations of values, of course, may be equal under a given interpretation, so that two different representations may denote the same value.

Object identity is a value type of particular importance (see previous subsection). By means of object references, objects can be shared and accessed from different contexts (property O4). On an implementation level, the possibility to share objects is an important concept for reuse and integration, but also a source of major headaches (side-effects through aliasing). No such concepts and problems exist for values (property V4).

Objects have an object state, which can described as a set of attributes. An attribute has a name and a value that is bound to the name at any point in time. An object implementation refers to an attribute

using its name. Accessing the value part of an attribute given its name must be directly supported by the programming language of a given implementation.

Chapter 4

Objects and Values in Modeling

In software development, we can distinguish at least three different types of models: domain analysis, system design, and implementation models. As we have illustrated, objects and values are not just implementation concepts defined by programming languages, but modeling concepts that can (and should!) be used in domain analysis and system design as well.

There is no fail-safe technique to decide whether some tangible or intangible concept is of a value or object type. The properties of objects and values, which we have listed above, characterize what it means for some concept to be an object or a value, after it has been decided to model it as such. The modeling decision itself is always a pragmatic one, driven by questions like: How do users handle this particular concept (analysis model)? How many instances are likely to exist (design model)? How heavyweight is its default representation (implementation model)?

In fact, different models may define a concept to be of different types. A good example is the concept of Address. Conceptually, an address is a value, and if a person moves, it is not that an address changes, but rather that the person moves to another address. Thus, in the analysis model, Address is a value type. However, an address is not a lightweight abstraction, so not only modeling Address as a value type, but also implementing it with full copy semantics, might cause performance penalties.

An equally tricky example is the concept of Collection. In principle, collection types like Set, Map, and Array are higher-level constructed value types. Implementation-wise, every object system we have seen implements them as objects.

While no hard rules exist, and every decision must be done pragmatically, our experience shows that value types are better reserved for lightweight abstractions.

4.1 Implementation techniques

Most object-oriented programming languages do not directly support the implementation of value types. One reason might be that they tend to have escaped the attention of the programming language

designer, because they were focusing on objects. Another reason, perhaps more important, is that there is no single best implementation of value types. For example, is not always appropriate to use copy semantics, because heavyweight values might slow down the system and memory consumption might get too high.

If an object-oriented programming language or system does not provide direct support for value types, they must be implemented as classes. Different implementation techniques can do this in different ways, serving different needs and different purposes. We can categorize the different implementation techniques along the following two dimensions:

How to ensure value semantics of an object. The implementation techniques of this dimension can be subdivided into two major categories: (1) Immutable objects (i.e., objects whose state cannot be changed). (2) Copy-On-Write objects, which provide a mechanism that ensures that the value object will be copied before a mutating operation is applied. This approach guarantees that two clients A and B of the same value object V cannot incidentally change each other's state through changes to the value object V. Thus, changes to value objects solely have an effect on the client that invoked the operation.

How to make best use of a specific programming language. The major differentiation criterion here is the method of how objects are created. Languages like Java and Smalltalk only support dynamic object creation (i.e., the object is allocated on the heap). C++ also supports the concept of static object creation. Static object creation means for clients that the value type is realized as an embedded object, and it means for operation arguments that they will be passed on the runtime stack rather than dynamically allocated on the heap. In the following sections we refer to heap allocated values as *dynamic values*, and to values created on the stack or embedded into other objects as *static values.*

This section reviews different implementation techniques to realize value types in object oriented systems. With each implementation technique, we provide a list of advantages and disadvantages in order to ease choosing the best one given a set of concrete requirements. The following sections are structured according to the two dimensions described above.

4.2 The Immutable Object approach

Immutable objects are a simple and effective implementation technique to realize value types in object oriented systems. An immutable object is an object whose state may not be changed, because is does not provide any operation to manipulate it, or (low level) operations, which try to change the object's state, will result in a runtime error. Immutable objects can be implemented either statically or dynamically.

The advantage of immutable objects is the guarantee of freedom from side-effects, and the clear implementation of the concept of value types. Additionally, in multi-threaded environments, no locking of the value object is needed, because the value never changes.

Its disadvantages are the object creation overhead, and increased memory consumption due to an increased number of objects. If an immutable object is created statically, (i.e., allocated on the runtime stack or embedded in the client object), an additional disadvantage is that polymorphism is lost [Str97]. If immutable objects are created dynamically, an additional disadvantage is that a garbage collector becomes mandatory.

Flyweights help resolve some of the disadvantages of the naive immutable object approach. A flyweight is a shared object with no extrinsic (i.e., context-dependent) state [GHJV95]. Consider the example of adding the monetary amount of "CHF 3.00" to the monetary amount of "CHF 5.00". Rather than directly creating a new value "CHF 8.00", the add operator of the value object "CHF 5.00" re-

quests the “CHF 8.00” value object from a value manager (flyweight factory in [GHJV95]). This manager might reuse an existing “CHF 8.00” object or choose to create a new one if it doesn’t yet exist.

Flyweights have the prime advantage of reducing memory consumption to the minimum. It is guaranteed that for every value, there is at maximum one value object that represents it. From this, another advantage follows: Comparison for equality of two value objects can be reduced to comparison for identity (two structurally equal value objects are guaranteed to be identical), so performance increases.

Using flyweights, creation of value objects becomes more expensive, because retrieving the value object requires at least one additional associative lookup. Particularly for complex value types (value types with further embedded value types), the calculation of the hash value and the comparison of two value objects by the value manager might cause a slow-down.

Also, immutability of value objects sometimes turns out to hinder optimal performance. Consider the following example: an associative collection (e.g. a Java hashtable) manages a list of immutable MonetaryAmount value objects. To increase each monetary amount by “CHF 5.00”, the following code sequence is typical for Java:

```
Hashtable aMonetaryAmountMap = new Hashtable();
...
MonetaryAmount oldValue = (MonetaryAmount)aMonetaryAmountMap.get(aKey);
aMonetaryAmountMap.put(aKey, oldValue.add(5));
```

The problem associated with the code sequence is the double associative lookup. First, to get the value from the map, and second, to write the value back into the map. It would be desirable to just write:

```
((MonetaryAmount) aMonetaryAmountMap.get(aKey)).add(5);
```

This code sequence manipulates the value in place, thereby omitting the second associate lookup to put the value back in place. This problem can be solved by either implementing special classes (e.g. collections, iterators, and converters), which know that they are dealing with values, or by realizing value types as normal objects. But then, there must be a mechanism which ensures that value objects, which are referenced more than once, will be copied before the manipulating function is called. Otherwise, value semantics will be lost.

4.2.1 The Copy-On-Write Approach

Copying the value object upon write access can take place in one of two places. First, we describe a solution where the copying responsibility is located at the client side. Then, we show how the Body/Handle idiom can be used to delegate the responsibility to the value itself.

4.2.2 Client-Side copy

The simplest solution is to impose the obligation to maintain value semantics on the client. Then the developer of a value type only needs to implement operations to copy the object, and the client must ensure that the value semantics are maintained. Typically this means making a copy before changing the value object. The client must know what it is doing and when to perform this copy.

The solution can be used both for static and dynamic object instantiation. Depending on the chosen method, the class must offer different operations. If the values are statically allocated (C++), the copy constructor and the assignment operator have to be implemented. If the values are dynamically allocated on the heap, they must provide a deep-copy operation.

The primary advantage of this approach is that it can be applied to classes, which have not been designed as value types from scratch. The only functionality the class must offer is a copy operation (see above). Its clear disadvantage is that it is not fail-safe but relies on a disciplined use by the client programmer. Nothing specific can be said about its performance. The memory consumption will be mediocre due to many redundant copies.

4.2.3 The Body/Handle idiom

The problem of changing the reference before the referenced object will be manipulated can be solved through an additional indirection. This indirection is typically implemented by using the Body/Handle idiom [Cop92]. This idiom suggests implementing an object in two halves, the body, and the handle. Clients always hold a reference to the handle, while the handles reference the body objects. The actual state of the value object is maintained by the body, with the handle only mediating the access to it. Normally, the Body/Handle idiom is combined with a reference counting mechanism. In this case the body carries a reference count, typically an integer, telling how many handle objects hold a pointer to it. If a mutating operation is called on the handle, and the reference count of the body is not equal to one, a new body is copied from the old one. Furthermore, the reference count of the old body is decreased by one, the reference count of the new body is set to one, and the body-pointer of the handle is set to the new body. Then the mutating operation is delegated to the new body.

To ensure that the handle cannot be referenced more than once (which obviously violates value semantics), it must be copied before it can be passed as an argument to an operation, or when the object holding a reference to it is duplicated. These situations are identical with the situations in which a value object without the body handle concept itself has to be duplicated. So the usage of the Body/Handle idiom is useful only if the copy process of the handle object can be automated. Therefore this solution is predominantly used in C++ where the handle can be implemented as a static object. The new operator of the handle class must be private to avoid the instantiation of dynamic handles. Additionally the constructor, copy constructor, destructor, and assignment operator must be implemented to manage the reference count mechanism correctly. We know of no constraints for the implementation of the body class.

The advantage of this solution is that it is well balanced. Both memory consumption, if the body is realized as a flyweight object, and the performance for read operations are optimal. Merely the performance for write operations is mediocre. In C++, this solution has the advantage that garbage collection can be implemented easily using the reference counting mechanism of the Body/Handle idiom. Using reference counting for implementing garbage collection is sufficient for lightweight values because no cyclic dependencies are allowed. An additional advantage is that the handle object can do the locking of the value (the body object) in multi-threaded environments. Thus, the client code does not need to take care of this. The Body/Handle idiom makes it furthermore possible to use statically allocated values polymorphically. The only prerequisite is that all handle objects must be of the same size. This is typically the case.

The major disadvantage of this approach is that references to value objects (for example in C++ `MonetaryAmount& rValue` or `MonetaryAmount* pValue`) must be forbidden by programming conventions (see section 6.1). Otherwise side-effects become possible. Furthermore the indirection of the Body/Handle idiom results in increased object access latency.

The following table summarizes the discussion about the different implementation techniques. The symbols ++, +, o, and - have the meaning optimal, sub-optimal, mediocre, and poor.

	Immutable objects (as flyweight)	Client-side	Body-Handle (body as flyweight)
clear realization of the concept	++	-	+
memory consumption	++	o	++
Performance to create a new value	exists as flyweight: + otherwise: o	o	+
Performace when using a value	++	++	read: ++ write: o
Usage in collections	o	++	++
Suggested programming language	Java ST (C++)	None	C++

Table 1: Summary of the different implementation techniques

4.3 Programming languages

Programming languages might provide direct support for value types. For example, in [KS95], Koenig and Stroustrup make an argument for the use of user-defined data structures. Their argument resem-

bles ours, except that they view the need for small data structures from a programming language perspective.

How might language support for value types look like? To the user, it would look like he or she is provided with first class user-defined value types, which can be handled much like integers and strings are handled naturally in most programming languages.

Another solution would be to enhance a language's type system so that it becomes possible to ensure the desired value properties. The developer provides an implementation and is sure that a client is unable to exploit the object properties of the representation. The type checker would treat any such attempt, e.g. the attempt to obtain a reference to the representation, as a compilation error. It depends on the programming language how much has to be done for this approach. The range of changes needed reaches from adding just one operator to introducing a new keyword *value (type)***.**

The implementation of such a feature might use any of the aforementioned approaches, most notably immutable objects, possibly provided as flyweights, and objects, with the copy mechanism provided by the objects themselves. The last approach can either be realized through the Body/Handle idiom or by the following mechanism: whenever a client calls a mutating operation on a value object, the object first copies itself. Next it replaces the reference to it hold by the client to the new copy. Then the original value object forwards the mutating operation to the copy. Please note that the needed language support goes beyond the `become:` language mechanism of Smalltalk. `become:` in Smalltalk changes all references to an object A to the reference of the object B, and, in some implementations, every reference to the object B to the reference of the object A (two-way become).

It would be desirable, if the programming language definition not only introduced the concept of value, but also lets developers provide implementation hints, which basic implementation technique is to be used (in particular with or without flyweights).

Chapter 5

Project Experiences

We report on our experiences with value types from a number of large object systems.

5.1 GeBOS

The GeBOS series of banking projects developed at RWG GmbH (Stuttgart, Germany) consists of more than 3500 classes. It strongly relies on the concept of value type. Value types were introduced after a first prototype for the investment business had been finished. The major reasons to introduce value types were to avoid side-effects and to simplify the mechanism for storing objects in relational databases. A domain value framework was developed which allows the simple addition of new domain specific value types to the system.

We chose Copy-On-Write as our implementation technique, considering it the most appropriate approach in C++, our implementation language. The implementation is a variant of the Body/Handle idiom combined with reference-counted garbage collection. Furthermore, we implement the body parts of frequently occurring values as flyweight objects.

Each domain value is able to give a description of itself. This is used to check the value for validity, to provide a string representation of it, etc. The state of a business objects (except their relations to other objects) is captured by lists of such values. This lets us provide features like editing or constraint checking in a generic way. Since its introduction the value framework has become more and more important as a foundation of other subsystems. We use primitive value types like integer only for the most basic operations like loop counters, but not for domain-specific computations. The last achievement was the development of a new legacy system wrapper on the central mainframe system, which receives and delivers its transaction data primarily as lists of domain values.

The GeBOS projects use a large number of different value types. We use general domain value types like amounts, strings, identifications; specific date-related value types like day, month, year, and time span; financial domain value types like monetary amount, account number, interest rate, stock quotes, and interest due dates. Etc.

5.2 Geo

The Geo system is a distributed software system developed at Ubilab, UBS (Union Bank of Switzerland). It is implemented in Java. Right from the beginning we have imposed a strict distinction between object types and value types, which we implement as immutable flyweight objects. We have modeled and implemented many concepts as value types: time and date, object identifiers, resource descriptions, any kind of symbol (for example, in the context of type descriptions), the whole exception hierarchy, and the full technical name hierarchy (file names, class names, package names, etc.). On average, our performance analysis shows some slight slowdown in performance, about 5%, due to the value management complexity. The number of objects has been reduced to 53% on average application runs. The overall memory footprint has decreased by about 10%. This unexpectedly meager reduction of the memory footprint is based on the large Java runtime system, which does not use value types the way we have described (with the notable exception of the String class).

In the Geo system, our primary example applications are of technical nature. One exception is a three-tier bibliography system, which introduces a number of value types specific to the bibliography domain. One set of examples are the field types (taken from the BibTeX definition) for fields in a bibliographic entry (the entries themselves are implemented as object types). Another set of examples are the layout token types used to control the output of our formatting algorithms for generating reference lists (like the one at the end of this paper). A final example is the set of symbol types for a regular expression parser.

5.3 KMU Desktop

The KMU Desktop system is a workplace system for account managers in the corporate banking division of UBS, developed by its IT division. Already in the conceptual phase of the project we made a clear distinction between object and value types. The KMU Desktop PC-based client is implemented in Smalltalk. It uses the immutable object approach for value types in general. For value types with a small cardinality, it combines the immutable object approach with flyweights. The flyweight factory is hidden in the implementation of our value type framework.

We provide most basic banking domain value types like currency, monetary amount, and interest rate. In addition, we provide banking domain specific adaptations of primitive value types like string and floating point number. Finally, we have introduced technical domain value types like object identifiers.

5.4 Summary

In all of these projects, we have made very good experiences with the distinction between objects and values. This includes all models: analysis, design, and implementation. We think that for large-scale business applications the concept of value type and its distinction from object type is key for understanding a system and defining its overall performance.

Chapter 6

Related work

We review the (implicit) use of value types in C++, Smalltalk, and Java, as well as in the proposed "Objects By Value" standard of CORBA and distributed systems in general.

6.1 User defined data types in C++

In [KS95] Koenig and Stroustrup state that "it was an explicit aim of C++ to support the definition and efficient use of such [i.e., small concrete] user-defined data types very well". Starting with the typical built-in value types (characters, integers, floating point numbers), they list several lightweight types as examples, which all turn out to be what we consider value types. This indicates that the authors acknowledge the importance of value type support. The solution they provide is to allow programmers to declare variables of user-defined types like variables of built-in types, e.g.:

```
int i;     // variable of type int (built-in integer)
Date d;    // variable of type Date (user-defined)
```

In this example, d is not a reference, but directly bound to an object of type Date. This is definitely an advantage compared with languages like Java where only references to user-defined types are possible.

Lets consider which of the properties of value types d fulfills. As we have seen on the implementation level, we deal with value representations that have a lifetime. So the best we can expect concerning timelessness (V2) is that we don't have to worry about the lifecycle of these representations. C++ fulfills this property: the representing object is created when d becomes visible and is destroyed when d goes out of scope.

Theoretically it depends on the interface design whether d is immutable (V3). Koenig and Stroustrup suggest to handle user-defined data types with object semantics. They provide the following example:

```
class Date {
public:
[...]
  Date& add_year( int n );        // add n years
  Date& add_month( int n );              // add n months
  Date& add_day( int n );         // add n days
};
```

These operations do not only modify the date, but also return a reference to the date object, thereby allowing side-effects.

It would be helpful to have a facility that prevents a value representation from being aliased. Such a feature would make it possible to implement value types whose operations are guaranteed to be side-effect free. A possible approach in the spirit of C++ is to introduce an `operator alias()`, which is called when a reference is obtained. For value types the operator could be declared private to prohibit aliasing. In the absence of this feature, one always has to adhere to programming conventions (see section 4.2.2).

We conclude that C++ was designed to support small non-polymorphic *object* types very well but lacks language support for pure value types.

6.2 Smalltalk

Value types also exist in Smalltalk. However, they can not be mapped directly to classes, but only to a concept called *literal constants* as defined in [GR89]. A literal constant in Smalltalk is a *description* of a constant object like a number or a string. There are five types of literal constants: numbers (e.g. 2, -30.4), characters (e.g. $a, $b), strings (e.g. ‘this’, ‘that’), symbols (e.g. #flagOn, #flagOff) and arrays (e.g. (1 2 3), (1 ‘food’ $s)). All these literal constants are implemented as immutable objects: they are normally protected by the virtual machine, sometimes marked read-only, and the creation of new objects (i.e., #new, #new:) is forbidden. However, this does not apply for the classes Array and String of which anyone can create new instances and manipulate them.

It is not a simple task to implement value types in a totally open and reflective system like Smalltalk. We see three ways to implement them:

In our custom-provided superclass of all value types, we can have an attribute called *isMarkedReadOnly*, which is tested by each instance variable setting operation of the value type.

We can test the current value of an attribute for #nil. This allows lazy initialization of value objects, which are composed of multiple value objects and/or basic Smalltalk objects. If the attribute is not #nil, this indicates that someone tries to set it for a second time and appropriate action can be taken.

In some Smalltalk implementations like IBM VisualAge [Smi94, IBM95], there is an undocumented private feature for marking an object read-only (#markReadOnly:). This is used to protect literal constants, and can be used to implement immutable objects.

Which solution to choose depends on the complexity of a value type. However there is *no way* in Smalltalk to fully protect an object from being manipulated.

6.3 Java

As a prominent programming language, which takes a rather hybrid approach, consider Java [AG96]. In Java, strings are instances of a class String, which is implemented as an immutable object with Body/Handle separation and flyweight body objects. Thus, it provides value semantics. Unfortunately, this has not been carried through to other value types. For example Date, Point, and Rectangle, even though uniquely qualified to be implemented as value types, provide object semantics, with all

its described problems (see section 2). As a consequence, programmers must enforce client-side value semantics, if they want to avoid these problems.

6.4 The proposed CORBA standard for “Objects By Value”

The need for passing parameters by value in distributed systems is obvious (see section 2). Therefore, the OMG issued a request for proposals to make this possible within the CORBA standard. This resulted in a joint submission [OMG98], which introduces the concept of value type. Only instances of value types can be passed by value. As in the past, instances of object types cannot be passed by value.

A value type definition starts with the new keyword *value.* It consists of the declaration of an interface and (optionally) a specification of the value representation. The representation specification allows the ORB to transfer the representation to another context, possibly implemented using another programming language. The new values are intended to be used as parameters but not as object attributes, because object state specifications are not part of the CORBA object model. The proposed CORBA value types provide full subtype polymorphism. Value types have the following properties:

- Value types are not CORBA object types: they inherit (implicitly) from the new type `CORBA::ValueBase` instead of `CORBA::Object.`
- Values don’t have an identity. When passed as parameters, a copy of the representation is created in the receiving context.
- Values don’t have an object lifecycle: they aren’t created nor destroyed. The client does not have to care about the creation and destruction of the representation copy.

It is not possible to create aliases to value instances on the interface level.

The only property that CORBA value types cannot ensure is immutability. This property is missing, because the IDL provides no means to specify const operations (i.e., operations that do not modify the instance). Concerning the C++ language mapping the authors considered to map all operations of value types to *const* member functions. However, this would have introduced incompatibilities with the existing object type mapping.

We consider some of the proposed features to be problematic. For example, it is allowed for a type to inherit both from a value type and an object type, resulting in unclear semantics. Also the language mappings were not designed to enforce a strict distinction between objects and values. On an implementation level, users of CORBA value types have to handle objects instead of values.

6.5 Other distributed systems

Many distributed systems have devised concepts to address the performance problems illustrated in section 2. From the simple immutable object approach, to design patterns like Flyweight, up to elaborate object movement and parameter passing schemes, a number of remedies are known.

As a representative of this group of systems, we pick Emerald [JHLB88]. Emerald provides direct support for object mobility in a number of different ways. Objects can be moved explicitly, using a set of primitives defined for this purpose. Objects may be clustered and always moved together. Objects

may be moved implicitly, triggered by a specific parameter-passing scheme like call-by-move or call-by-visit.

Many of the arguments made for object mobility functionality and their specific implementations can be explained by the distinction of object and value types. Immutable objects are an excellent means for increasing performance of lightweight objects in distributed and concurrent systems. Call-by-visit is an excellent means for reducing the frequency of cross-process calls for embedded objects. While all these concepts stand on their own, they frequently can be explained well using value types. Object mobility and modeling with value types complement each other.

Chapter 7

Conclusions

Objects and values are the primary modeling concepts of modern software systems. Yet, in object-oriented systems, values have gained no sufficient attention, to the extent that so-called “pure” object-oriented programming languages and systems deny that they build on values as much as they build on objects.

Our experiences with large object systems indicate that much can be gained from a clear understanding of values in object systems and their appropriate application, both as a domain modeling and as an implementation concept. General benefits, like freedom from side-effects, are complemented by technical benefits, like significant performance gains in distribution, concurrency, and persistence.

We present the key properties of value types in the context of object-oriented systems modeling. We analyze the design and implementation space of value types in modern mainstream object-oriented programming languages, and present implementation techniques. We discuss how proper modeling with value types can increase system understanding and performance in a wide area of technical domains. Finally, we present our experiences from a number of projects, which have made a clear distinction between object and value types.

Based on the analysis of value type properties, the consequences of their application in analysis, design, and implementation on system understanding and performance, and our project experiences, we conclude that value types are fundamental to object system design and implementation. They significantly help with issues of largeness by omitting many of the problems, a naive object-oriented approach causes.

References

AG96 Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

Cop92 James O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.

Cun95 Ward Cunningham. “The CHECKS Pattern Language of Information Integrity.” In Pattern Languages of Program Design. Edited by James O. Coplien and Douglas C. Schmidt. Addison-Wesley, 1995. Page 145-156.

GHJV95 Erich Gamma. Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

GR89 Adele Goldberg, and David Robson. *Smalltalk-80 The Language.* Addison Wesley, 1989.

IBM95 International Business Machines Corporation. *IBM Smalltalk User’s Guide, Version 3, Release 0.* International Business Machines Corporation, 1995.

JLHB90 Eric Jul, Henry Levy, Norman Hutchinson and Andrew Black. “Fine-Grained Mobility in the Emerald System.” Readings in Object-Oriented Databases. Edited by Stanley B. Zdonik and David Maier. Morgan Kaufman Publishers, 1990. 317-328.

KS95 Andrew Koenig and Bjarne Stroustrup. “Foundations for Native C++ Styles.” Software-Practice and Experience 25, S4 (December 1995). Page 45-86.

Mac82 B. J. MacLennan. “Values and Objects in Programming Languages.” *ACM SIGPLAN Notices* 17, 12 (December 1982). Page 70-79.

OMG95 Object Management Group. *CORBAservices: Common Object Services Specification*. Framingham, MA: OMG, 1995.

OMG98 Object Management Group. *Objects by Value.* OMG document id orbos/98-01-01. Framingham, MA: OMG, 1998

Smi94 David N. Smith. *IBM Smalltalk: The Language.* Benjamin/Cummings Publishing Company, Inc., 1994.

Str97 Bjarne Stroustrup. *The C++ Programming Language.* 3rd Edition, Addison-Wesley, 1997.

WJ95 Roel Wieringa and Wiebrien de Jonge. “Object Identifiers, Keys, and Surrogates: Object Identifiers Revisited.” *Theory and Practice of Object Systems* 1 (2): 101-114.

Ubilab Technical Reports

94.6.1	Maffeis S, Bischofberger WR, Mätzel K-U: GTS: *A Generic Multicast Transport Service*
94.9.1	Bischofberger WR, Kofler T, Mätzel K-U, Schäffer B: *Computer Supported Cooperative Software Engineering with Beyond-Sniff*
94.9.2	Bäumer D, Bischofberger WR, Lichter H, Schneider-Hufschmitdt M, Sedlmeier-Scholz V, Züllighoven H: *Prototyping von Benutzungsoberflächen*
94.10.1	Steiger P, Ansel Suter B: *Minnelli Schlussbericht*
94.10.2	Levy N, Hornstein T: *Text-to-Speech Technology: A Survey of German Speech Synthesis Systems*
95.6.1	Riehle D: *Muster am Beispiel der Werkzeug und Material Metapher*
95.7.1	Riehle D, Schäffer B, Schnyder M: *Design and Implementation of a Smalltalk Framework based on the Tools and Materials Metaphor*
97.1.1	Riehle D: *A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose*
97.3.1	Brudermann R: *GeoTransporter—Entwurf und Implementierung eines Objekttransportes für das Geo-System*
97.6.1	Mätzel K L, Schnorf P: *Dynamic Component Adaptation*
97.7.1	Barja M L: *A Comparative Evaluation of OODBMSs*
98.5.1	Marsura P, Riehle D: *Design and Implementation of the Java Any Framework*

Paper copies of Ubilab technical reports can be ordered from the mailing address on the first page or by e-mail from its author using the scheme firstname.lastname@ubs.com. Most reports can also be obtained as PostScript files via WWW (http://www.ubs.com/ubilab).

Abstract

Objects and values are fundamental yet complementary concepts of software system modeling and implementation. However, in the context of large object systems, modeling and implementing value types has received no sufficient attention. Our experiences show that proper understanding of value types can ease programming and improve performance of such systems significantly. In this paper, we discuss the impact of value types on object-oriented system design, implementation, and performance. We discuss several implementation techniques that let us introduce value types into systems implemented in mainstream object-oriented programming languages.