Abstract Data and System Design: from Bits to Component Objects

Ulf Schünemann
Department of Computer Science
Memorial University of Newfoundland (Canada)
ulf@cs.mun.ca

Current address:
Institut für Informatik,
Albert-Ludwigs-Universität Freiburg (Germany)
schuenem@informatik.uni-freiburg.de

Technical Report 170
28^{th} of January, 2002

Abstract

Data in computer systems is an abstraction in form of software entities at several levels: bits, values of the programming language, instances of abstract data implementation modules, and dynamically composed "abstract" objects. Each level has a corresponding notion of data, represented by aggregations of entities at the level below. The separation of users and implementation of data at each level increases system modularity. Frontal breaches of abstractions by misinterpretation and lateral breaches by access to the parts are a threat to modularity and sound modular reasoning in terms of high level data entities. Type systems and modular encapsulation protect values and instances of implementation modules, but not the objects of object-oriented design, black-box reuse and component-based software. To achieve object encapsulation, several programming disciplines with differing in flexibility have been proposed. They work by controling the references which expose the object's representation (alias control).

Keywords: data abstraction, representation, modularity, object abstraction, aggregation, composition, composite object, component object, black box reuse, representation exposure, alias control.

Abstract Data and System Design: from Bits to Component Objects

Ulf Schünemann; ulf@cs.mun.ca; Memorial University of Newfoundland; January 28, 2002

1 Introduction

Data in computer systems is an abstraction existing in form of different kinds of software entities (figure 1):

1. bits, with which computers work.
2. values of the programming language.
3. instances of abstract data implementation modules.
4. dynamically composed "abstract" objects, as in object-oriented and component-based software.

Over the last 60 years, the forefront of software production and of programming language design shifted from bits towards abstract objects: the trend is towards "higher" concepts of data, which are abstractions from "lower" ones. For this abstraction to be sound, it has to be protected from low level interference. But why are higher level data entities so interesting?

First, higher level data entities are closer to extra-software ("real") world concepts which humans use commonly for understanding and solving problems in a given domain, and farther away from a solution executable on real computer hardware. Thinking at higher levels allows, as Brooks [Bro87:Sil] expressed it, to **focus on the essence** of a software solution to a problem; and it allows to reduce the "accidental" complexity, which comes from the technology (computers, programming languages, libraries) with which the solution is produced. Strongly typed languages, e.g., encapsulate the bit-level so that programmers can safely ignore it. Programming in higher-level languages was "[s]urely the most powerful stroke for software productivity, reliability, and simplicity" [Bro87:Sil].

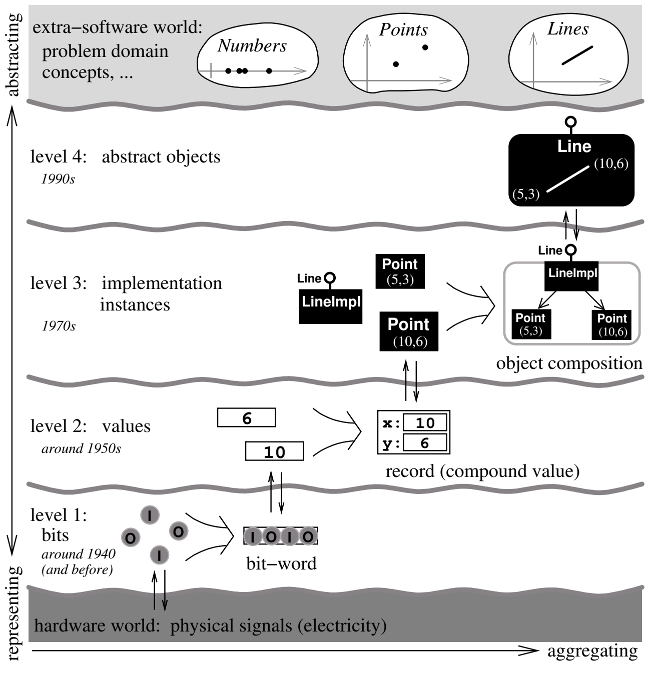
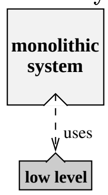


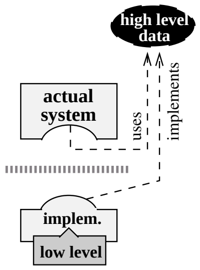
Figure 1: Software levels

Second, higher level data entities are **more powerful**: simulating one at a lower level requires, in general, to combine *several* lower entities. In the end, a complete solution has to bring any data into bit form to make it processable by a computer. — Software with a machine to execute would only be a hypothetical solution.

The third motivation is **modularity**: Within a system's life-time, lower level decisions are more likely to be revised than higher ones [Mar96:DIP]. Changes in low level data details should require adaption only to code accessing this data. In monolithic systems this access code is not isolated, so that low level changes may entail a cascade of changes in the system, or make it break at unexpected places. This is *bad system design* [Mar96:DIP].



If one identifies low level access that serves to implement higher level data then it can be disentangled and separated by an abstraction boundary from the rest of the system. Both parts then depend on the same high level data concept, but not on each other. This decouples the high level essence of the solution, the actual system or "identity of the application" [Mar96:DIP], from the details of its low level implementation: The data implementation can be reused in different systems, and the system can be reused on top of different implementations, which contain the low level changes.



Correctness of high level thinking about real systems and independence of system components are the benefits of abstraction. Guaranteeing them requires protecting the abstraction boundaries, for which automatic techniques exist at levels one to three. But for the composed objects of black-box reuse and component systems there is no guarantee, only guidelines. The protection concept "object encapsulation" is well-understood and supported *only* for instances of implementation class modules. Preservation of the central object-oriented abstraction, the object, is left to programmers' intuition and self-discipline by current languages, design methods and specification techniques:

The big lie of object-oriented programming is that objects provide encapsulation [Hog91:Isl].

The following sections will each discuss for one level (1) a notion of data, (2) a way of representing it at the previous level, (3) the benefit for system design, and (4) the abstraction boundary's breach and protection.

2 Information Bits

Information is what remains after one abstracts from the material aspects of physical reality. [Res87:IlRe]

The *smallest* unit of information is in genetics a letter of the genetic alphabet (A, C, G, T), in spoken language the phoneme [Lyo89:EML], and, in *information theory* since around 1940, the bit [Res87:IlRe]. A **bit** is the amount of information reducing the number of alternatives by half, i.e., the answer to an "optimal binary question" [Bis95:SuB], formally an element of $\mathsf{B} = \{\mathsf{I}, \mathsf{O}\}$. Information is conceptually independent of meaning [Bis95:SuB], but can be assigned meaning by interpretation: Triplets of genetic letters mean amino acids in protein synthesis. The phoneme combination k+u+d means, in English, the word COULD. The bits I+O+I+O, interpreted as unsigned integer, means ten (see also §3). Such combinations of information units are the carriers of meaning, the individual unit serves merely as a discriminator [Lyo89:EML].

Manifestation bit ↔ electricity. Information needs the manifestation as a stable physical quantity, a "signal" [Bis95:SuB], to be usable: Genetic letters are represented by nucleotide bases, phonemes by sounds, and bits typically by electricity: a voltage of, e.g., three volts is bit I, no voltage is bit O. The bitwise electrical scheme of representation, used in telegraphy since the mid-19th century, superseded decimal mechanical representation in computing hardware in the 1940s.[1]

A computer M works with millions of logically different bits $b \in Bits_M$ whose values $\sigma(b) \in \mathsf{B}$ are represented in corresponding circuits $rep(b)$. The single bit, however, is not the computer's unit of storage and manipulation. All bits occur in groups of n, called **bit-words**, whether in memory cells or in computer operations.[2] A cell C combines a set $ext(C) = \{b_1, \ldots, b_n\}$ of bits, whose values $\sigma(b_1) \cdots \sigma(b_n)$ combine to C's value $\sigma(C)$. A (virtual) memory address $a \in Addr$ is the identifier for a cell $C = Cell[a]$. There is no genuine identifier for a specific bit in a computer.

Binary software. Without software, each computation to be automatized would require to build a different system S_0, or to rebuild an old one by rewiring or the like. Proper computing systems are separated by an abstraction boundary into a binary program S_1 (software world), and the computer hardware M (physical world) executing S_1: The same hardware M can be reused without changes for many applications simply by feeding it new data or programs S_0'. And programmers and computer scientists are saved from physical intricacies (race conditions, heat effects, etc.); the circuit "place" $rep(b)$ of a bit $b \in Bits_M$ may switch at runtime to exploit special fast *cache memory* circuits; and software is freely reusable on different machines M' with the same binary interface (e.g., with improved chip design and a new microcode simulation of the machine language).

Breach? Computers are engineered to make the hardware:software boundary a virtually unbreakable barrier: From the immaterial world of software, details of the physical working cannot be observed. (A loophole might be the possible overheating and destruction of chip circuits through program execution. This breaks abstraction since it depends on the chip design, and resulting malfunctions are software observable.)

3 Values in Programms

A notion of value. What the developers of the binary electronic computer, like its ancestors, had in mind was not the manipulation of bits, but the automatization of mathematical calculations (hence "computer"). Computers directly support calculations within some subsets of the numbers, the numerical values. Calculation in these limited sets, characterized by a range and a resolution,[3] is studied by *numerical analysis* [Rei92:Num]. During the 1950s, the notion of value and computation was generalized to include Boolean values, characters (of sets evolving out of telegraph codes), memory addresses, etc., and their compounds (c.f. §4.2). These types of values can still be understood from their non-software counterparts; a systematic approach to data arose only in the '70s (see §4.1).

Encoding value ↔ bit-word. In order to be processable in a binary computer, numerical and other values have to be *encoded* (represented) in bits, or rather, as explained before, in bit-words, i.e., aggregations of

[1] Its advantage is the relative independence from physical inaccuracies, and that the physical information manipulators can be made simple, reliable, and ever faster and smaller (relays, vacuum tubes, transistors, gates, ...).

[2] Zuse's Z3 of 1941 [Thu97:Z3] had 22-bit words, and the Ferranti Mark I of 1951 [Tur51:Man] 40-bit words. Nowadays, 32-bit words are common; 8-bit words are also called bytes.

[3] Zuse's Z3 of 1941 [Thu97:Z3], for instance, calculated with floating point numbers up to nearly 10^{19} (ten million trillion), where the distance from one positive number x to the next is ca. 0.012% of x, and the number next to 0 is ca. $5 \cdot 10^{-20}$.

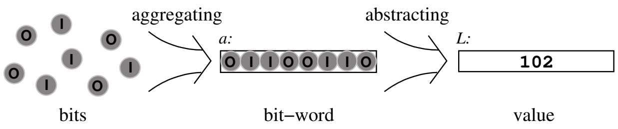


Figure 2: From bits to value

bits. As an example, the idea for representing an integer like hundred-two by a 8-bit word is to write it in base-2 number notation, replace *0*s by Os and *1*s by Is, and add Os to the left: $rep(102) = rep(\mathit{1100110}_2) =$ OIIOOIIO. The reverse process is to interpret bit-words as integers, as shown in figure 2: $int($OIIOOIIO$) = \mathit{1100110}_2 = 102$.

More generally, the essence of the relationship between the world of values of type τ and the world of n-bit words is captured by an τ:$\mathbb{B}^n$-kind **representation scheme** $\langle rep, int \rangle$. Mapping $rep : \tau \to \mathbb{B}^n$ defines how values $v \in \tau$ are represented as bit-words $rep(v)$, and $int : \mathbb{B}^n \nrightarrow \tau$ defines how bit-words w are interpreted as τ values $int(w)$. int can be partial ("$\nrightarrow$") as explained below. All represented values have to be interpretable ($\mathrm{im}(rep) \subseteq \mathrm{dom}(int)$) so that the value can be recovered ($int(rep(v)) = v$ for all $v \in \tau$).

Interpretation is an act in the human mind. Which interpretation a *computer* (or program) is said to be using depends on the operations it applies to the bit-word, based on a man-made classification of operations: integer operations mean interpretation as integer, logic operations interpret as Boolean, and so on. Consequently, operation applications will be crucial for questions of the abstraction boundary (see further below).

A problem, recurring analogously at other levels, is that for some type τ of values there may be less different values than bit combinations within a word.[4] Then either the interpretation of unneeded combinations is left undefined (int_τ is partial), or several bit-words represent the same value (int_τ is not injective). In the former case, the behavior of τ operations applied to uninterpretable bit-words is undefined; it may hang up the program or corrupt the data. In the latter case, the implementation of τ operations is made more complicated: For instance, machine provided bit-word comparison is insufficient for testing equi*valence*.

Values are contained in **variables** $V = Var[l]$, identified by a *location* $l \in Loc$.[5] Locations are also called *l-values* since they are the *values* to which *l*eft hand sides of assignments evaluate. A variable V's representation $rep(V)$ is a memory cell. V's value $\sigma(V)$ is the interpretation $int(\sigma(rep(V)))$ of the bit-word stored in V's representation. Mapping variables $Var[l]$ to cells

[4] For example, neither Booleans values (true or false), nor ASCII-67's 128 [Jen01:ASC], EBCDIC's 127 [Win83:Char] or Unicode's 49194 characters [Uni00:Unicode] match with an 8-bit byte's 256 or a 16-bit word's 65536 possible combinations.

[5] "Variable" is also said to the zero, one or more syntactic names x, y, etc., which can be given to V in a high-level program.

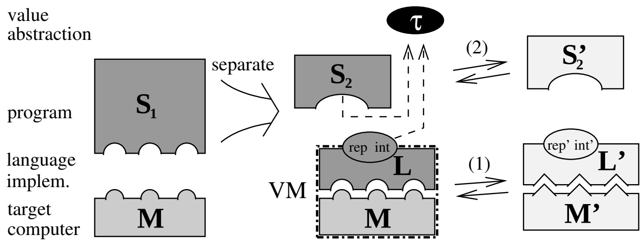


Figure 3: High-level language system architecture

$rep(Var[l]) = Cell[a]$ is trivial if the cell's address is used as the variable's location: $l = a$. rep may be more complex in systems with type or dimension descriptors.

Software in high-level languages. In binary software S_1, values are hard coded in their bit form—representation schemes cannot be changed separately from the program. "Automatic programming" in the 1950s allowed writing programs in a not directly machine executable form [Bac78:Fort]: Such programs could contain symbolic references to library functions, and be expressed in terms of not necessarily machine supported values, variables, and operations on them. This development culminated in the first **high-level programming languages**, like Fortran I (1957), Lisp (1958), Algol (1958/60), and Cobol (1960) [Lou94:PSp].

Making a high-level program executable requires to extend the dichotomy of program and hardware M by a third party, the **language implementation** L (figure 3). It includes foremost a compiler program to translate "source" program S_2 to "target" machine M's language (or an interpreter program to execute S_2 directly), and also a runtime library and a link editor to resolve library references. L, together with M, the executer of the compiled program, forms a system for executing S_2; $L + M$ is a "virtual" machine VM (dash-dotted line) programmable in the high-level language.

The decision how to represent S_2's values and variables in M lies in compiler L: L translates literal values to bit-words, variables to memory cells, and value operations to bit-level operations. Ideally, source program S_2 will be decoupled from L by the abstraction boundary of the language's datatypes τ. This makes the program's meaning independent of particular representation schemes, thus freeing the programmer from "accidental complexity", and allowing to focus on complexity inherent in S_2 [Bro87:Sil]. Also it enables safe reuse of S_2, and of L in different settings:

1. *VM substitutivity for program portability*: Replacing the representation scheme for τ values does not affect S_2. In other words, S_2 can be reused unchanged with any other virtual machine $L' + M'$ for the language, where computer M' can have a completely different architecture (depicted as triangular plugs between L' and M' instead of half-circles).

2. *VM integrity for VM reuse*: It depends entirely on compiler L which bit-words occur where τ values are expected. Programs do not interfere in the compiler's maintaining of the integrity of the representation scheme and thus of the stability of the τ abstraction. Hence the virtual machine $L+M$ of program S_2 can be safely reused for program S_2', even if some bit-words are uninterpreted.

Strong typing. If programs are not known to respect the abstraction boundary of τ data then programs and compilers cannot be exchanged independently: First, safely porting a program S_2 to a compiler with a different representation scheme will require to find operation applications which break the abstraction, and adapt them to the new scheme. Second, in each new program S_2' for compiler L, all abstraction breaking applications have to be found and checked: If, at a place where τ values are expected, they produce bit-words without interpretation, then τ's implementation in L has to be adapted to give some well-defined interpretation to such bit-words.

Two forms of violating the abstraction of data can be distinguished (at level 2, as well as at other levels):

In a frontal breach, the computer (the program) *misinterprets* τ value v as value d of some non-τ type σ, or vice versa. That is, it interprets $rep_\tau(v)$ as $int_\sigma(rep_\tau(v))$ instead of $int_\tau(rep_\tau(v)) = v$, or $rep_\sigma(d)$ as $int_\tau(rep_\sigma(d))$ instead of $int_\sigma(rep_\sigma(d)) = d$. In other words, it applies non-τ operations to $rep_\tau(v)$, or τ operations to $rep_\sigma(d)$. For example, a character value c could be misinterpreted as integer $i = int_{\texttt{int}}(rep_{\texttt{char}}(c))$ so that $64 < i < 91$ is the test whether c is upper case.

This works fine as long as $rep_{\texttt{char}}$ assigns ASCII codes to characters. Porting a program with such a test to a representation of characters by IBM's EBCDIC codes [Win83:Char] requires adapting the check to $(192 < i < 202) \vee (208 < i < 218) \vee (224 < i < 234)$. This dependency could be avoided by doing the testing with an operation `isupper : char → bool` that is part of the language. On the other hand, the source program might introduce "characters" with codes beyond ASCII (codes 0 to 127). Applying `char` operations like `ispupper` to a code like 223 would misinterpret it as a character: $int_{\texttt{char}}(rep_{\texttt{int}}(223))$. The typical compiler's implementation of `isupper` by lookup into an array of 128 Boolean flags would need to be extended to the new codes, which effectively extends the domain of $int_{\texttt{char}}$.

If the language has only one type of values then a wrong way of interpreting bit-words is not possible. For example, the machine languages of Zuse's Z3 of 1941 [Thu97:Z3] has only floating point operations and that of Aiken's Mark I of 1944 [OR99:Aiken] had only integer operations to manipulate bit-words. Algol (1958/60) had many different types of data. Its designers of Algol did not trust the programmers to avoid all breaches. Instead programmers were protected from breaches through a **strongly typed language**: Variables have to be annotated with the type of value as which bit-words in the corresponding memory cell are to be interpreted [SG96:SAr]. Based on this, the compiler determines statically to which type of values expressions will evaluate. The payoff of writing annotations without computational effect is that a *type error* is flagged where operations on a value deviate from an interpretation according to this type.[6] This effectively keeps separate the values of different types.[7]

Lateral breaches can occur where the representation $rep_\tau(v)$ has embedded the representation $rep_\sigma(d)$ of another value d. For example, the representation of floating point numbers contains the representation of an integer exponent, of an unsigned integer mantissa, and of the number's sign. Implementing a programming language's floating point operations on an integer-only machine requires letting integer operations manipulate exponents and mantissas.

A lateral breach of τ value v's abstraction is the application of operations to some representation $rep_\sigma(d)$ embedded in $rep_\tau(v)$ without that these application are part of implementing τ operations on v. v's abstraction is broken, even if the applied operations are σ operations and thus do not break d's abstraction.

If a frontal breach cannot be used to gain access to a part of the representation, then only the compiler (and language library) L has access to the representation. Operations implemented in L not normally make calls to source program S_2 ("up-calls"). In this case, the access will be contained within L, so that lateral breaches can easily be avoided.

4 Data Abstraction and Classes

Once the usefulness of computers was demonstrated in the 1950s, new application areas for computers were found, which were not only numeric. The notion of computing was generalized in the 1960s from calculating with numbers to an increasing variety of application-specific types of data. In programming, finding the right data gained importance besides finding the right algorithm. "In the late 1960s, good programmers shared an intuition about software development: If you get the data structures right, the effort will make development of the rest of the program much easier" [SG96:SAr].

But exactly what is meant by data? [ASS96:SICP]

[6] Checking for misinterpreting operation applications can also be deferred to runtime, as in Lisp, and a variable's type can also be inferred without annotation from its initialization value, as in ML [Mi+97:SML].

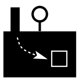
[7] Common holes in a strict separation regime are the construct of a union type (other than ML-like datatypes), and reinterpreting type casts which do not convert the bit-level representation.

4.1 The General Notion of Data

Liskov, among others, worked on "figuring out what the concept was" in the early seventies [Lis92:CLU]. She studied data in an abstract fashion, irrespective of how and at which level it is represented. Making use of any kind of data requires some *constructor* operations to create it and some *accessor* operations to do something with it. Consequently, the abstract notion of some data, the **data abstraction**, is captured by characterizing the corresponding operations, i.e., by specifying name, signature and behavior (result and side-effects) of all constructors and accessors [ASS96:SICP].

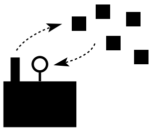
The three main styles of data abstraction shall be explained with dates of the Gregorian calendar, with focus on the "class style." A data abstraction will be depicted as a large black box with a chimney symbol for constructors, a lollipop symbol for accessors, and square boxes for data entities.

An individual style data abstraction characterizes one single, possibly mutable, data entity like the date «*today*». The date «*today*» in the software world has



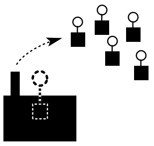
a constructor for initializing it to the current day in the real world. And it has *observer* accessors determine «*today*»'s year, month and day, and a *mutator* accessor, called `Advance`, advances «*today*» to the next day. (An individual may be used as *prototype* for creating further, similar entities [Un⁺91:PWC], but this will not be discussed here.)

An abstract datatype, or ADT style data abstraction characterizes a set (*type*) of immutable data entities by operations on and between them: a construc-



tor delivers dates, and observers determine their different aspects and the date of the day before or after a given date. This is a most natural way of understanding *values* (§3). However, how *variables* of a certain type change cannot be characterized.

The class style somehow combines both styles. At each invocation, the constructor creates a new mutable data entity, called *instance* of the class. Each class's



instances are a new *type* (or "class" [Boo94:OOAD]) of data. Each instance is an *individual* with accessors on itself and between itself and others. A class style data abstraction characterizes the constructors and all these accessors. This allows talking about advancing a date by a mutator, versus determining next and previous dates by an observer.

Names and signatures of operations of the class `Date` of dates can be specified by the following declarations in a Java- or C++-like syntax:

```
Date(int d, int m, int y);
void    Advance();
int     dayOfMonth();
int     month();
int     year();
int     dayOfYear();
boolean sameday(Date d2);
Date    next();
Date    prev();
```

Here and in the following, constructors are named after the class, and names with small initials are used for observers, and capital initials for mutators. Each declaration gives, in this order, the operation's result type, name and, in parentheses, the types and names of its parameters (name and result type of constructors are merged to one).

In class style, accessors do not take the instance to access as parameter. Instead, each instance has its own accessors. They are referred to with the "genitive notation" [Bi⁺79:Simula]: If a is a `Date` instance, then a`.Advance` is "a's `Advance`" mutator, and invocation a`.Advance()` advances a by a day.

State model. A complete data abstraction has to go beyond declarations in programming languages and characterize also the operation's *behavior*. The behavior of an accessor applied to (several) instances can depend on the constructor calls (which constructor, which parameters) which created them, on the history of mutator calls which affected them, and on the environment at the time of the application. One way of describing this in a precise, yet abstract way are model-based axiomatic specifications, like Larch/C++ [Lea99:LC++]: Each instance is assumed to have a discrete *state*, expressed by an explicit model. The behavior is specified in dependency of, and as effect on the states of the involved instances. An operation's *postcondition* relates its result and the instances' states at the end of a call to the parameters and instances' state at application time.

Class `Date`'s specification is given in figure 4 using Java syntax (set in typewriter font) augmented by specification clauses in pseudo syntax. Java keyword `final` is only relevant for inheritance (see §4.5). Functions in sans-serif font are auxiliary to the specification, not part of the `Date` data abstraction. isdate will be explained below. The complete specification with all auxiliary functions is given in appendix A.

The three spec clauses introduce d, m, and y as model variables. They are, as triple $\langle d, m, y\rangle$, the specification's explicit model of a `Date` instance's state. In this case, the model variables coincide with the results of the accessors `dayOfMonth`, `month`, and `year`. A model variable x is simpler data used for the specification of some more complex data D. Hence, x is instance of some data abstraction X, e.g. `int`, with its own specific operations, like addition, less-than, etc., specified with some model variables $x_1, x_2, \ldots$. These variables are a kind of "data group" [Lei98:DG] factored out of D's

```
data abstraction class Date {
  spec d  = final int dayOfMonth();
  spec m = final int month();
  spec y  = final int year();
  Date(int d, int m, int y)
          requires isdate(d,m,y)
          ensures d = d ∧ m = m ∧ y = y;
  void Advance()
          ensures d  = old dayOfNext(d,m,y)
                ∧ m = old monthOfNext(d,m,y)
                ∧ y  = old yearOfNext(d,m,y);
  ... }
```

Figure 4: Specification of class `Date`

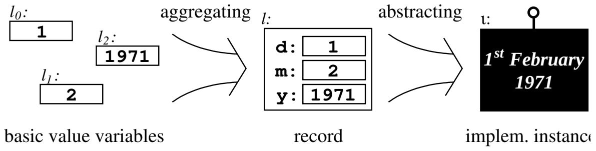


Figure 5: From basic values to implementation instance

specification and modeled as X data.[8] Polygons, for instance, can be specified using points $\alpha_{1,\ldots,n}$ as model variables for (grouping the coordinates of) the n corner points. Reifying data groups allows reusing and factoring data abstractions, and seems to be the way of choice for "polylithic" [Ja$^+$94:OOSE] entities with inherent non-shallow structure, like trees or stick figures.

Specification of behavior. A requires clause specified for an operation states the conditions ("preconditions") only under which calling it makes sense w.r.t. the abstract data entity. The constructor has the precondition that the given parameters refer to a legal date. This is expressed through predicate $\mathsf{isdate}(d, m, y) :\Leftrightarrow$
$0 < m \leq 12 \wedge 0 < d \leq \mathsf{daysInMonth}(y, m)$.

Ensures clauses state an operation's postcondition. In order to relate state before and after the operation's invocation. names and expressions quoted by old stand for the value they have at the invocation's begin, whereas unquoted names and expressions stand for the value they have at the invocation's end. For example, each `Advance` changes a date's day, month and year aspects to those of the day following it before the change.

The clarification of the operations' behavior allows further reasoning about data entities. For example, one might identify model states for which the specified behavior is identical. Also general properties of the state spaces can be deduced, like properties that are always true (invariants) or change monotonically (history invariants) [LW94:BehNot]. Dates can only go "forward," not backward, in time since this is what the only mutator `Advance` does (the "history rule" [LW94:BehNot]). All `Date` instances that can be encountered in a system are legal since the constructor starts with legal dates and `Advance` never leads to illegal dates ("datatype induction" [LW94:BehNot]). That is, $\mathsf{isdate}(d,m,y)$ is an (implicit) *model invariant*, an invariant property of `Date` instances' model state.

[8]The "extension" of a data group [Lei98:DG] in subtypes is the specialization of x to a subtype with additional model variables.

4.2 Implementing Data $\leftrightarrow$ Record

Data abstraction is a powerful approach for dealing with data in an abstract way. Also the computer's bits and the language's values can be characterized this way. But the design of a *general purpose* programming language can never anticipate and include *all* interesting types of data. Instead, programmers extend the language by introducing their own, **user-defined data**. Since the bit-level has been abstracted away in (strongly typed) languages, values are the lowest form of data in programs. They are as concrete as programmers can get, whereas they obtain user-defined data only by abstraction from values. Consequently, in programs the language's values are the *concrete* data, whereas user-defined data is called *abstract* data [Lis92:CLU].

Aggregating. W.r.t. data, "every powerful language" provides basic values, constructs to aggregate several values, and ways to deal with such an aggregation as one value again (**compound value**) [ASS96:SICP]. *Heterogeneous* aggregation constructs like tuples (Lisp) or records (Algol) "enable us to implement the concrete level of our data abstraction" [ASS96:SICP]. (Homogeneous aggregations like arrays are less interesting here.) Dates could be represented simply be translating the aspects day, month and year of the state model to integer variables V_0, V_1 and V_2 (figure 5). With the record construct these variables can be given the respective labels `d`, `m` and `y` to yield the record V with these variables as components ("fields"), i.e., $ext(V) = \{V_0, V_1, V_2\}$.

The language provides generic operations on (mutable) compound values (creation, copy in and out of variables and procedures, comparison, selection of components). A generic way for compilers to find bit-level representations of compound values is to combine them from the representation of their components: Record V is represented in memory by the cells $rep(V) = \{rep(V_0), rep(V_1), rep(V_2)\}$. Normally, the compound's representation is one contiguous area of memory: V_0, V_1 and V_2 are represented at addresses a, $a + 4$ and $a + 8$, respectively for some a (assuming 4 byte integers).

Abstracting. If the user-defined data isomorphic to the compound, one may argue that aggregating suffices. For example, complex numbers $a + i \cdot b$ "are" records `{re=`a`, im=`b`}` (in ML notation [Mi$^+$97:SML]), and 3-D points, vectors, or triples $\langle x, y, z\rangle$ "are" records `{x=`x`, y=`y`, z=`z`}`. Records like `{d=1, m=2, y=1971}` might already look like dates. But they are mere combinations of three integers, which is not the same: Neither are any three integers a date (c.f. the isdate invariant, §4.1), nor are dates necessarily integer triples: `{y=1971, d=32}` "is" the 1^{st} February (the 32^{nd} day) of 1971 with the same right as `{d=1, m=2, y=1971}`. Also values of derived types in Ada [ISO95:Ada] or datatypes in ML (e.g., `date(1,2,1971)`, or `date(1971,32)`) are merely tagged isomorphic copies of other (compound) values in the programming language.

All the data entities *in* a language are basic values, compounds or their derivates. Really new, application-specific data entities and types with special properties can only be added by the programmer at a level *above* the values provided by the language. That is, after aggregation with the language's constructs, an additional, intellectual act is needed (figure 5): One has to **abstract** from the compound's concrete properties, to think of it as a date, and treat it as such. Figuratively, an abstraction barrier is drawn around compound L (white box), which, from the outside, looks like a date o (black box). The (concrete) compound inside the box is said to represent the (abstract) date.

A record $V = rep(o)$ representing a user-defined data instance o is called *instance record*. The fields $ext(V)$ of V are called o's *instance variables* $Rep(o) = ext(V)$. The data entity obtained by abstraction from a compound shall be called *implementation instance*, as opposed to a "language defined" value, and user-defined data and data abstraction in general (c.f. §5). An *object identifier* $\iota \in OID$ is a value which identifies an implementation instance $o = Inst[\iota]$. o's model state $\sigma(o)$ is the (user-defined) interpretation $int(\sigma(V))$ of the value of record V. The simplest way to relate implementation instances $o = Inst[\iota]$ with their instance records $rep(o) = V = Var[l]$ is to chose as o's object identifier $\iota = l$ in the first place.

Implementation. In order to produce usable user-defined data abstractions, the programmer must not only *design* representation schemes, but also *implement* the abstractions' operations. This is *possible* in any language with function definitions (Lisp, Algol, etc.). User-defined data with a one-to-one representation by compounds is directly *supported* by special programming language constructs, often called "modules" [Str87:WIOOP]: They declare a new type of data, how it is represented, and define all the operations. In some languages, types introduced this way do not have the semantics of (first-class) values but have implicit reference semantics. In this case, implementation instances cannot themselves be aggregated again to represent other implementation instances, but only their object identifiers (which are values). This yields linked data structures, which will be discussed in §5.2.

The first object-oriented language Simula (1967) [Bi+79:Simula], intended for system simulations, was the first to support (class style) data abstractions by its **class** construct. A Simula class is similar, both syntactically and semantically, to a procedure declaration with local procedures and variables. Whereas variables defined in the procedure body are stack-allocated on invocation, the instance variables in the class body are heap-allocated on instantiation.

From the 1970s on, constructs were designed more specifically for the then better understood concept of data abstraction [Lou94:PSp]: classes in Smalltalk (developed 1972 to 1980); instantiable "modules" with type fields in Euclid (1977); for implementing ADTs, the **clusters** of CLU (1974 to 1977), and, 1990, the **structures** of Standard ML (evolving out of constructs in CLEAR (1977) and HOPE (1980) [Mi+97:SML]); and, the **module** proper with statically allocated variables for individual data abstractions in Mesa (1979), Modula-2 (1982) and Ada (1978 to 1983).[9] Finally, the **class module** construct of C++ (1983) [ISO98:C++] and Java (1994) [GJS00:Java] augments the class construct by module variables and operations ("static members").

Each of these module constructs, however, can also be used for other things than data abstractions (c.f. footnote 10); inferring the data abstraction from a given module is not always straightforward (c.f. §4.5).

An implementation of class `Date` in Java where each model variable of the specification (§4.1) is simply represented by a corresponding private instance variable can be found in appendix A. More instructive is a variant implementing a `Date:{x,y:int}`-kind representation scheme with no instance variable for months, where the days in a *year* are counted in instance variable `d`.

```
class Date {
 private int d, y;

 public final int dayOfMonth() {return dayOf(y,d);}
 public final int month()  { return monthOf(y,d); }
 public final int year()   { return y; }

 public Date(int dd, int mm, int yy)
            { y=yy; d=dd+daysInPrevMonths(yy,mm);}
 public void Advance()
            { d=d+1;
              if(d > daysInYear(y)) {d=1; y=y+1;}
            }
 public int dayOfYear()  { return d; }
 public boolean sameday(Date d2)
               { return d==d2.d && y==d2.y; }
 public Date next()
   { Date d2 = new Date(dayOfMonth(),month(),y);
     d2.Advance();
     return d2; }
```

[9] Modules allow but do not support implementing types or classes of data [Str87:WIOOP, Pre97:KSF].

```
public Date prev()
  { return d==1? new Date(31,12,y-1)
         : new Date(dayOf(y,d-1),monthOf(y,d-1),y);
  }
// auxiliary functions
... }
```

`private` and `public` are "access modifiers" (c.f. §4.4). Simply put, `public` identifies the operations as parts and `private` identifies the instance variables as not parts of the data abstraction. The definition of auxiliary functions `daysInYear`, `monthOf`, and `dayOf` can be found in appendix A.

Programs with data abstraction constructs not only define program semantics at the level of variables. A module as above also defines that a `Date` instance o is represented by a record $V = int(\{\texttt{y=}\sigma(V_0)\texttt{, d=}\sigma(V_1)\texttt{\}})$. Thus one knows where the abstraction boundaries are, so that program semantics at the implementation instance level can be inferred, i.e., it is also defined within the program. This *actual* semantics can be checked against the *specified* behavior (§4.1), for example to reveal the actually implemented representation scheme:
$\sigma(o) = int(\sigma(V)) = int(\{\texttt{y=}\sigma(V_0)\texttt{, d=}\sigma(V_1)\})$
such that $\sigma(o).d \;= \texttt{dayOf}(\sigma(V_0), \sigma(V_1))$
$\sigma(o).m = \texttt{monthOf}(\sigma(V_0), \sigma(V_1))$
$\sigma(o).y \;= \sigma(V_0)$

Properties specified in terms of model variables can now be translated to instance variables. *Model* invariant $\mathsf{isdate}(d,m,y)$, e.g., requires $\texttt{0} < \texttt{d} \leq \texttt{daysInYear(y)}$ as (level-2) *representation* invariant of correct implementations based on this representation scheme.

4.3 Modular Software

In 1968, McIlroy had the vision of an industry for reusable *software modules*, and in 1972, Parnas defined a module as *hiding a design decision* [Pre97:KSF]. However, it remained nebulous how modules were to be identified, until the notion of data abstraction was taken as guidance [Lis92:CLU]. That is, the parts of a data abstraction implementation formed one module, thus hiding the design decision about the representation. In the following, each style of data abstraction inspired a *programming paradigm* for identifying modules [Str87:WIOOP].

Modules I which implement data abstractions D extend the virtual high-level language machine $L + M$ (dash-dotted line) by a vocabulary of data abstractions [SG96:SAr]. Therefore user-defined data abstractions are a good basis for structuring a high-level program S_2: The client part S_3 of S_2, which deals with a data abstraction D per se, can be separated from D's implementation I defined elsewhere in the program. On the other hand, modules allow to structure the programming language into a standard library of data abstractions (implemented as ordinary modules) and a simple core language (implemented by the compiler).[10]

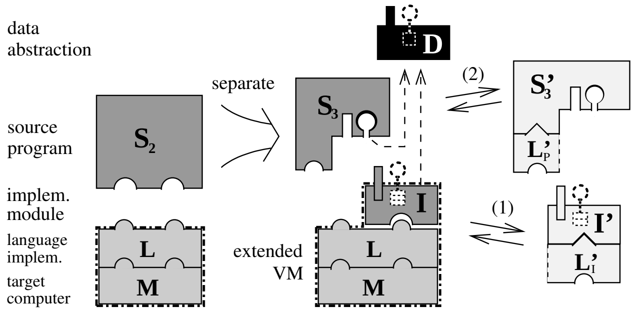


Figure 6: Modular system architecture

Modules supersede procedures as the main organizational unit [Lis92:CLU] in development, testing, reuse and libraries. In development projects, libraries and the market can be searched for a module implementing a data abstraction D to be used in design. However, more likely than modules which are exactly as wanted will be modules which implement D only partially. Such modules could be adapted to save an implementation of D from scratch. But modifying the source code is often not an option, especially in case of modules in compiled form. The solution is the object-oriented concept of **inheritance** (subclassing) in Simula 1967: A new class (subclass, or derived class) can be defined by "differential programming" [Un+91:PWC] of the "delta changes" [Pre97:KSF] through which it differs from an untouched existing class (base-class, or superclass). Classes are "open" for adaption and extension, even if their source code is "closed" (the "open-closed principle" [Mey88:OOSC]). More on inheritance will be said in §4.5.

The use of abstraction. The specification of a data abstraction D lays down the "official" operations for creating and accessing D-data and their "official" properties. In Meyer's **design by contract** [Mey88:OOSC], the specification is seen as a contract between the module I implementing D and the client S_3: It obligates S_3 to use only the official D-operations in working with D-data, and to use them only with the respective precondition satisfied. In return I guarantees the data's model invariant and the operations' postconditions.

The program S_3 has to abandon any representational view of the data [Mee94:TaxDT], i.e., respect its abstraction boundary. More precisely, S_3 has to fulfill its part of the contract and rely only on *official* properties of D-data and D-operations. S_3 must not make assump-

[10] Only very few modules of standard libraries of languages like Ada, ML, C++, Java, and Smalltalk [ISO95:Ada, GR97:SMLBL, ISO98:C++, Sun00:JavaAPI, GR83:ST] do *not* implement a data abstraction. They are collections of constants, of type meta information, or of additional operations on existing types.

tions about, nor interfere with I's representation of D-data and implementation of D-operations. It would be "disastrous" to assume a particular representation — "This applies as much to second-order representational assumptions (e.g. a complex value as an ordered pair of real values, or an array as a contiguous block of values) as it does to first-order assumptions such as bit patterns" [Mee94:TaxDT].

This way, S_3 does not directly depend on I's (accidental) properties, but both, S_3 and I, depend on D which defines the official (essential) properties. "This effective decoupling of modules is indispensable for the development of large programs, in particular, if modules are developed by different people" [Wir83:Mod2]. S_3 and I can now each be reused in different contexts:

1. *Module substitutivity for evolution*: Evolving (maintaining, correcting, improving, or adapting) a program here means substituting a new version I' for an old version I of the module implementing D.[11] Substitutivity means that this has no effect on the rest S_3 of the program (including I's subclasses).[12] For example, the behavior of data abstraction `Date` in §4.1 is *uniquely* specified, so that effects and observations made through `Date` operations are the same for all correct implementations of `Date`. Since S_3 is not affected, it is reusable without changes with any implementation variant. This allows isolated revisions of the representation scheme for D data.
2. *Module integrity for reuse*: Client systems do not interfere in the module's maintenance of invariants over its representation scheme, whether implied by the model invariant, or independent of it. The module alone provides for the stability of the implemented data abstraction. Hence a module version I implementing data D for system S_3 can always be reused safely for other systems S'_3 needing D data.

4.4 Modular Encapsulation

If client S_3 of module I violates the abstraction boundary, all sorts of terrible things may happen. It will not help if S_3 or I worked perfectly before with a different version of, respectively, I or S_3:

Let us assume an accounting system S_3 which checks variable `today` of type `Date` to issue paychecks to all employees every first day in a month. Instead of using operation `dayOfMonth`, S_3 might access instance variable `today.d` of `today` directly to test it against 1. This (frontal) breach of the abstraction boundary will not surface as long as a three-integer implementation I_3 of class `Date` (see appendix A) is used where `d` in fact contains the day of the month. But there will be a big surprise when `Date` is updated to the two-integer implementation I_2 defined in §4.2: Suddenly there will be just one single payment in the entire year!

In another accounting system S'_3 a date's `d` variable might be increased in steps between 27 and 31 to jump directly from the first day of one month to the next in a two-integer implementation I_2. There is no violation of specification (model invariant, etc.) since the same can be achieved by a loop of `Advance` calls. But if three-integer implementation I_3, which worked perfectly in system S_3, is transfered to S'_3, it will start to misbehave: For I_3, numbers in `d` above the numbers of days in current month `m` are an unexpected violation of its *representation* invariant. Since I_3 does not check against illegal dates, it can happen, e.g., that operation `dayOfYear` returns numbers above the number of days in the year. For S'_3, this is an unexpected violation of a property derivable from the `Date` specification.

Either way, since the client program S_3 did not expect what it sees it may itself show completely unplanned behavior, that could result in corruption of persistent data or could be exploited to break security. In such situations it is easy for the developers of S_3, and of I to blame each other of improper programming (violation of contract [Mey88:OOSC]). This is a problem, especially with commercial module libraries sold in compiled form.

Encapsulation. If programming teams in a software project do not trust each other to stay on their side of the abstraction boundary they can use the programming language to impose appropriate restrictions. Such a limitation of representation access to the implementing module is called modular encapsulation. More generally, encapsulation is a technique for ensuring that certain design decisions remain hidden in a module.

Strong typing originally prevented frontal breaches at the level of the language's values (§3). This is insufficient for user-defined data since, in the above example, the use of `today.d` does not misinterpret its type. This is the situation in Simula (1967) [Bi+79:Simula], which had no encapsulation. But the separation of types of values can be extended to the types of the implementation instances added to the language at the level above values. Then date `today` is *not* of a record type with field `d`, but simply of type `Date`, so that direct access `today.d` is caught as a misinterpretation. This leaves clients R_3 to access dates only through `Date` operations.

[11] Class model `Date` used only concrete values. However, a module implementing some data abstraction D may itself be client of data abstractions D_i by using D_i values as parameters, results, local or instance variables, or by being a subclass of D_i. Substituting a new module version I' for I actually means replacing I by I' *and* some modules implementing the D_i used by I' (as far as such modules are not already part of S_3).

[12] Of course, different versions may have different time and space behavior. But the same kind of effect can occur at a lower level through a change of compiler or computer. In case of platform-specific versions it may indeed be the *purpose* to change the program's non data-related behavior in subtle ways, e.g., to confirm to the platform's standard look&feel.

Lateral breach of encapsulation through pointers to variables:
Let us assume, our Java-based language to be extended by C++'s pointers to variables. Then one could write a global (static) procedure `swap(ptr1,ptr2)` for swapping the values in the integer variables pointed to by `ptr1` and `ptr2`, respectively. Based on `swap` a corresponding `Swap` operation for `Date` objects could be introduced ("&x" yields the pointer to variable x):

```
class Date { ...
   public void Swap(Date d2) { swap(&y, &d2.y); swap(&d, &d2.d); }
}
```

An implementation of `swap` might store the pointers in global (static) variables `gptr1` and `gptr2`, respectively, e.g. to make use of some highly efficient function `gswap` which swaps the values of what these pointers point to:

```
   void swap(int *ptr1, int *ptr2) { gptr1 = ptr1; gptr2 = ptr2; gswap(); }
```

This way, at the end of calling d_1.Swap(d_2), anybody can obtain from `gptr1` and `gptr2` pointers to d_1's and d_2's instance variable `d`. Through these pointers d_1's and d_2's representation can be accessed anytime without the use of `Date` operations. The typical specification formalisms would not help detecting this breach of encapsulation: A specification of `swap` would include a so-called `modifies` clause to express that, besides `*ptr1` and `*ptr2`, also `gptr1` and `gptr2` may change by invoking `swap`. But this does not make visible the danger to encapsulation that comes with this version of `swap`, as opposed to the harmless classical one.

```
   void swap(int *ptr1, int *ptr2)
       modifies *ptr1, *ptr2, gptr1, gptr2
       ensures *ptr1 = old *ptr2 ∧ *ptr2 = old *ptr1;
```

But since `Date` is *user*-defined, some part of the program, `Date`'s implementation I, needs to be granted access to the representation of dates. For type checking it needs to be known which variables (and operations) may only be used within the module, and which operations (and variables, c.f. §4.5) belong to the data abstraction. A general approach are annotations specifying the scope of a declaration in the module. Java's `private` declarations, e.g., have module scope. They are not known outside of the class, and thus do not belong to the data abstraction. Hence, client program S_3's misinterpreting access to `today.d` will be caught as access to (private) variable `d` which is not in scope. All non-private members are visible somewhere outside of the class and therefore become part of the data abstraction embodied by the class. `public` declarations, e.g., have global scope. Java's other kinds of scopes will not be considered here.

Lateral breaches are more complicated: Instead of accessing a date's instance variable `d` directly, as in `today.d`, it could also be accessed through a *pointer* to variable `today.d`. A pointer to `today.d` can be obtained, e.g., by applying the C++ operator "`&`" to it. Since only module `Date` can access `today.d` directly to apply "`&`", all pointers to `today.d` must ultimately originate from module `Date`. The box on this page describes how such pointers may subvert encapsulation.

A clear-cut solution to lateral breaches is not to have operator "`&`" in the first place. Useful procedures like `swap` could still be defined utilizing the concepts of parameter passing *by reference* as with Pascal's `var` parameters [Lou94:PSp], or *by copy-in/out* as with CORBA's `inout` parameters [OMG00:CORBA]. These concepts are less general than pointers, but they have the advantage of being encapsulation safe: In the latter case, instance variables are accessed only at the start and end of the procedure's invocation—the external code has no access to instance variables. This *containment of the representation* to the module is a stricter form of encapsulation.

In the other case, the procedure does access instance variables (disguised as `var` parameters), but only as part the module's implementation of the data abstraction's operations so that it is not a breach. Since `var` parameters are only usable in (indirectly) called procedures, the module retains *control over the representation* by choosing which procedures to call. What external procedures might do with the instance variables is part of their semantics and thus covered by inductive reasoning. Containment and control of representation will be encountered again in the discussion of encapsulation at the next software level (§5.4).

Modular verification. Reasoning about the program in terms of abstract data (§4.1) should carry over to the actual behavior of the program. One would like to verify this by comparing only the implementation module I against the specification of the data abstraction D. However, such a modular verification "work[s] only when the [abstract] type's implementation is encapsulated" [Lis92:CLU]. Otherwise there could be a modification to the representation by the outside unlike one ever caused by code in I. "In essence, having a language enforce encapsulation means that the compiler proves a global property of a program; given this proof, the rest of the reasoning can be localized" [Lis92:CLU].

4.5 Module > Data Abstraction ?

The module construct in most languages is more general than one-to-one implementations of data abstractions. Some differences shall briefly be mentioned:

Non-private instance variables. An instance variables `x` of type τ might be declared `public`, or otherwise non-`private`. This can be explained as defining a *model* variable x and a pair `void` $\mathsf{Set}_x(\tau\, y)$ and τ $\mathsf{get}_x()$ of accessors for setting and returning x's value. As any model variable, x, and thus `x`, might be constrained by model (history) invariants. Preserving them is the shared responsibility of data users and data implementation. Operation Set_x, however, will frustrate any attempts to constrain `x` by *representation* invariants.

Implicit accessors. A variables private to a module still allow certain ways of observing and mutating an instance o which are not explicit as observer or mutator operations of o. Operation `Swap` in §4.4, e.g., means two mutators for date o: o.`Swap`(q) *and* q.`Swap`(o) for some date q. Also operations in an "inner class," which is lexically nested and thus a part of module `Date`, can "freely refer to the instance variables" of a `Date` instance [GJS00:Java]. All such implicit accessors have to be taken into considerations for correct reasoning.

Inheritance is a language mechanism to derive a new class module, the subclass, from an existing one, the base-class. The subclass is like the base-class, with three exceptions: It alone defines the constructors of its instances, although based on base-class constructors. It can extend the base-class's set of instance variables and accessors by its own. And it can "override" some of the base-class's accessors, i.e., provide its own implementation for them. Operations declared `final` in the base-class cannot be overridden, in Java. The base-class's instance variables and non-overridden accessors are said to be "inherited" by the subclass.

The overriding of accessors m in subclasses causes the **fragile**, or **instable base-class problem** [Pre97:KSF]: In an instance of the subclass, all calls of its own m accessor in its code (self-calls) are redirected to the subclass's version of m. Even calls within the inherited operations are redirected. This is put to use in the *Template Method* design pattern [Ga+95:DP]. But in general the effect of overriding on inherited operations is difficult to anticipate both by the programmer of the overriding subclass and by that of the base-class.

Such bi-directional coupling between classes destroys substitutivity and integrity. Protecting the abstraction boundary through encapsulation does not help since this is *not an issue of data abstraction*: Through inheritance the base-class module is not used as a data abstraction but for reusing and adapting the abstraction's *implementation*. The interface a base-class offers to subclasses is not only characterized by *what* data abstraction it implements, but also by the *how*. Without a specification of this interface, verifying that a subclass implements its data abstraction may require looking into base-class modules. In this sense, one could say "inheritance breaks encapsulation" [Sny86:EncInh].

Polymophism. In object-oriented languages, instances of subclasses can be used in place of instances of base-classes. That is, the subclass instances must satisfy the properties of the data abstraction D implemented by the base-class I. Is it possible to verify in module `Date` alone that its instances *and* all instances of its subclasses will satisfy the date specification? Note that without self-calls there is no instable base-class problem.

Wouldn't the introduction of new constructors (base-cases of the induction) and additional or overriding mutators (induction steps) in subclasses invalidate the verification? Each new constructor automatically calls a constructor in module `Date` to construct the inherited parts of the representation. And encapsulation ensures that any operation defined in a subclass can only read or modify inherited *private* instance variables by calling inherited accessors. That is, each new constructor's and mutator's behavior w.r.t. the `Date` data abstraction can be explained in terms of old ones. Hence subclass-based polymorphism is safe by the "extension-map" approach [LW94:BehNot].

For example, a subclass `Date2` of `Date` might *specify* an operation `Back` for switching the `Date` model state $\langle d, m, y\rangle$ to the *previous* day. But the `Date2` implementation cannot access the inherited instance variables `d` and `y` to modify them accordingly—`Back` cannot be implemented. Hence it remains a history invariant that no date can (be observed to) go backwards in time.

The only escape for the subclass would be to introduce its own instance variables `d2` and `y2`, which it can modify freely, and override operations so that they use these new variables instead of the old ones. In particular, operation `year` in `Date2` would return `y2`, i.e., model variable y is now represented by `y2`, not `y` any more. This loop-hole was closed in the example by making `final` all operations which implement `Date`'s model variables. A consequence is that the definition of a model variable's representation cannot be left open and deferred to subclasses.

5 Objects and Components

The previous section discussed instances of class-style data abstractions implemented by the language's class modules. Such *implementation class instances* are objects in the object-oriented sense, which are special in three ways: Implementation class instances

1. represent *data*, together with operations on it – they are "intelligent data objects" [ASS96:SICP].
2. are defined in their class and superclass modules with a static number of instance variables fitting into a fixed-sized chunk of memory – they are "Von Neumann objects" [Rum97:Myths].
3. are classified, or typed, by data abstractions (classes), whose constructors created them (possibly indirectly through a subclass constructor).

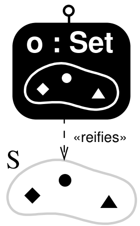
5.1 The General Notion of Object

Objects in the general sense are characterized solely by their **interface** of accessor operations which they offer as services to clients:

> "An object is an identifiable, encapsulated entity that provides one or more services that can be requested by a client." [OMG00:CORBA]

This establishes objects as abstractions in their own right, which shall be called **abstract objects** for emphasis. In object-oriented design, they are the "high-level objects." The abstract object concept in its full generality is at the core of *component architectures*, where it is called "component object" (see also below). A "component" is the packaging of the bit-level representation of the component object with an operation interface standardized at the bit-level [Hop00:Com]. And a "software component" in *component-based software* is an abstract object's compiled implementation (§5.3).

For example, an object o, the "set object", reifies a set S of objects as a software entity of its own: Whereas set S is characterized by which objects its elements are, the set *object* o is characterized by accessors to test an object's membership in the set, and to insert and remove objects from the set. Such an interface can be declared in object-oriented languages as a *fully abstract class*, i.e., a class without any implementation, which is called *interface* in Java, UML, COM and CORBA.




```
interface Set {
    boolean has(Object);
    void    Insert(Object);
    void    Remove(Object);
}
```

The type `Object` of the operations' parameters is Java's type of references to arbitrary objects. With a specification similar to that in §4.1, the behavior of set objects can be described precisely:

```
object type interface Set {
  spec has = boolean has(Object);
  void Insert(Object x)
       ensures ∀Object y• has(y) = (old has(y) ∨ y=x);
  void Remove(Object x)
       ensures ∀Object y• has(y) = (old has(y) ∧ y≠x);
}
```

The current state of a set object is accurately characterized by the membership predicate. Thus `Set`'s model variable is the mapping has : `Object` $\rightarrow$ `boolean`, implemented as operation `has`. result in postconditions stands for the value returned by the operation, and $\forall$ quantifies over all objects y. For simplicity, no care is taken to exclude `null` references. `Set` objects are actually subsets of the union of the set of (references to) objects and {`null`}.

Object $\gneq$ intelligent data object. The notion of object is more universal than that of data with its operations: An object can also be the invoker of operations of any other object in the system. Moreover, "all objects are conceptually concurrent" [Rum94:ComBB], as in Simula. Besides data this allows to model as objects active agents [Bi+79:Simula], communication protocols [Ga+95:DP], control structures [GR83:ST], etc.

State models are good for the specification of objects' data behavior, but they are not suited for expressing communication behavior. The corresponding analog to data abstraction could be called "**communicator abstraction**." Data and communication aspects together make each object a computing unit of its own. For computation in-the-large, the objects in the system cooperate through communications. Even *data* objects may require for their implementation by several implementation instances the use of internal communication (§5.2).

The concept of an object was developed out of the **programming language** community. From the very beginning [Bi+79:Simula], an object was data *and* communication. But on the other hand, in the programming language context, an object's data side never was more than an instance record (Von Neumann object). Nevertheless, object-oriented programming in the 1980s showed that such objects were useful enough for combining and encapsulating data and behavior, constructing data and communication structures (c.f. §5.2), and modeling non-software entities this way.

Object $\gneq$ Von Neumann object. The notion of objects as mutable data (and of data abstraction) is free from an encapsulated instance record's bounding of storage capacity at creation time:[13] A class's instance has for storage only the statically fixed number of instance variables declared in the implementation modules of that class and its superclasses. (Even Java's not class-defined array objects have a fixed number of (nameless) variables [GJS00:Java].)

Object-oriented software development methods, which mushroomed around 1990, integrated concepts from object-oriented programming languages, interface-based design and conceptual, or semantic, modeling [Ja+94:OOSE]. The programming languages' narrowing to Von Neumann objects and related conceptual constraints [Rum97:Myths] were overcome by conceptualizing objects independent of implementation modules and instance records. Such objects' "internal implementation is completely unconstrained" [MD95:COM], and their **representations are dynamically extendible open systems**.

[13] Some non-object-oriented languages, like ML, provide for recursive datatypes without using reference semantics. Values of these types are compounds of an unbounded number of variables.

During software development one deals with objects before their implementation determined. Objects treated as primitive at the time of their initial "discovery" may prove to have too complex a functionality for one class instance. This is obvious for component objects which can be "as large as whole applications such as Excel or PowerPoint" [Hop00:Com]. `Set` objects above, or CORBA's *collection objects* [OMG00:OCS] are assumed to "remember" any number of (references to) other objects, something which exceeds the storage capacity of any instance record.

Consequently, complex objects are successively decomposed into simpler objects during software design (§5.3), down to implementation class instances. The number of simple objects from which an object is composed may vary during the object's lifetime and is unbounded. That a combination of several objects can represent another object is a new idea. Before object-oriented design, structures of implementation instances were only understood to represent entities outside of the software world, like polygons and trees [Bi+79:Simula], collections and processes [GR83:ST], and so on.

Object $\gneq$ data abstraction. An object type is a more abstract characterization of objects (data entities) than data abstraction by the omission of constructors. This allows each part of the program to define its own, independent implementation; objects "provide a decentralized form of data abstraction" [Rey94:DT/PD]. One type of object can have multiple implementations, and new implementations can be added to the system at any time [LW94:BehNot]. An object type has an **extendible set of implementations**.

This aspect of objects is ideal for open, potentially distributed, loosely coupled systems, and hence became of interest in the second half of the 1990s with the raise of **component interoperability architectures** like COM, CORBA, and JavaBeans [MD95:COM, OMG00:CORBA, Ham97:JB]. Component systems are open to dynamic addition of new objects of any implementation. In CORBA, e.g., an "object broker" functions as linker between service requesting and service providing objects (c.f. §5.3). Because components are bit-level entities, the components in a system can have been programmed in different languages [Hop00:Com].

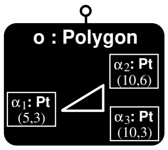
In Java, implementations take the form of concrete subclasses of an interface. (Note that interfaces do not give any hint of the representation.) The accessors defined in interface τ together with the constructors of all τ-implementing classes currently in the system are the data abstraction corresponding to object type τ.

On the other hand, this openness invalidates reasoning about (history) invariants of objects of a type τ using datatype induction or the history rule [LW94:BehNot]: First, omitting (the complete list of) *constructors* leaves us without a base case: For example, type `IDate` of abstract date objects could be specified simply by extracting the specification of accessors and model variables from the specification of the `Date` class data abstraction (§4.1). But without knowing the specifications of potential constructors it cannot be concluded that all `IDate` objects start their being with day, month and year aspects that combine to a legal date: isdate(d,m,y) is no model invariant for `IDate` objects!

Second, subtypes can introduce new *mutators* for changing the τ's model state in new ways, leaving us with an incomplete induction step. For example, subtype `IDate2` of `IDate` could introduce a mutator `Back` for switching to the *previous* day, and obviously implementations of `IDate2` can be given. Thus, it would be unsound to conclude from the `IDate` specification that `IDate` dates only go forward in time. (Remember from §4.5, that a sub*class* `Date2` of *class* `Date` might declare `Back`, but it cannot be implemented.)

The standard way of compensating for this lack is to specify explicitly the (history) invariants of τ objects [LW94:BehNot]. Such invariants are constraints on all constructors and mutators in any implementation of τ and of τ's subtypes: The constructors have to establish the invariants, and the mutators' changes to the state must preserve them and match the history invariants.

A polymorphic object [Ja+94:OOSE], as opposed to a monolithic `Set`, has its state modeled in form of other objects' state. Such objects, *model parts*,



are akin to model variables (called component fields and reference fields, respectively, in Fresco specifications [Wil92:FMOOP]). For instance, a `Polygon` object's corners (data groups of coordinates, c.f. §4.1) can be modeled as objects $\alpha_{1,\ldots,n}$ of an appropriate type `Pt`. The interface of `Polygons` can analogously be simplified by not offering operations for accessing the single corner coordinates, but by structuring it into operations `corner` and `SetCorner` for observing and mutating the corners [BC87:PartHier, VMO99:ESCO, HLS00:ObInt]. Having these operations means that for each model part α of the specification there will have to be a reification x at runtime, e.g., to return it from `corner`.[14]

```
interface Polygon {
    Pt    corner(int);
    void  setCorner(int, Pt);
    ... }
```

`corner` and `setCorner` may look like returning `Pt` objects and taking them as parameters, but in Java they actually return and take *references* ("reference semantics"). `SetCorner(`k`,`x`)` may make x the new (reification of) α_k, and `corner(`k`)` may return a reference

[14]The compound message selectors of [BC87:PartHier] would allow a `Polygon` to simulate the reification of its model parts.

to (the current reification of) α_k. Then mutating x will change corner α_k, and vice versa. If this "identity exchange" is the operations' purpose it should be specified explicitly in their postconditions [Wil92:FMOOP]. In the other extreme, the passed `Pt` objects may be no more than containers for transporting the coordinate data groups into and out of the `Polygon`. In this "transporter exchange," only `Pt` objects' state counts, not their identity. Clients may obtain different `Pt` object from repeated calls of `corner(k)`, without that this counts as a change of o's state. In general it may be left unspecified how the object x passed to `SetCorner` or returned from `corner` and o's corner α co-evolve, i.e., for how long changes to α can be observed through x, and for how long mutations to x imply changes to α.

5.2 Decomposing Abstract Object $\leftrightarrow$ Object Composition

Implementing an abstract object o like `Set`, requires one implementation instance r which provides a method for each operation of o. Let r's class be `SetImpl`. r will have difficulties implementing the expected storage behavior of `Set` operations. Since single implementation instances are not powerful enough, we apply the same technique as before: We aggregate several of them and draw an abstraction boundary around them: Storage of an unbounded number of references is achieved by using more or less instances of a simple class-style data abstraction `Node`. Its state model has to include a variable el for a reference to an object. The reference to each element in the set can now be stored as el in a different `Node` instance n_i. One `SetImpl` instance and a set of `Node` instances can represent `Set` objects (left side of figure 9). That is, the envisioned representation scheme is of kind $\texttt{Set} : (\texttt{SetImpl} \times 2^{\texttt{Node}})$ with

$int(\langle R, \{N_1, \ldots, N_k\}\rangle) = S \in \texttt{Set}$ such that

$S.has = \lambda x. \exists i = 1, \ldots, k. N_i.el = x$

Aggregating. How do objects become aggregated? It is not by some special construct, like with bit-words and compound values. Also objects in a system do not form aggregations merely because they are instances of the same class, or belong to the same system layer. Which objects are aggregated depends on the *references* between them. Since the network of references between objects (the "object graph") is not fixed statically, aggregation based on references can grow and shrink dynamically as needed during execution. This will be essential for adding new storage objects to the representation of an abstract set object whenever storage space is needed for added set elements.

With references we entered the realm of (dynamic mutable) *data structures*. Classically, they are aggregations of concrete values. In object-oriented languages like Java, however, all compound values are encapsulated in implementation instances. Thus, our data structures are actually *object* structures whose nodes are communicating abstract data entities (implementation class instances) instead of passive concrete values.

In a *concrete* aggregation A the aggregated objects themselves have references to each other. Although these references are defined in the objects' classes, the decision which structure to form may lay outside of the class. For example, consider instances of a class `LRNode` with references named **`left`** and **`right`**. Such nodes can be linked to aggregations called double linked lists (A_1 in figure 7), double linked ring (A_2), binary tree (A_3), or a lattice (A_4).[15]

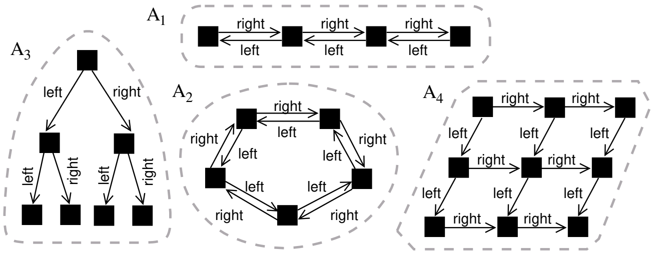


Figure 7: Concrete aggregations

Treat as one. References are a new kind of aggregation "glue:" Data or object structures are neither concrete values defined by record or tuple constructs, nor implementation instances defined by module constructs. Without such special constructs, how can the programmer treat aggregated objects *as a unit*, i.e., as an aggregation A which can be named and operated upon, and this even while it is empty or before all its elements have been created? "An aggregate is a union of several objects, and the union as such is often represented by an object of its own" [Ja+94:OOSE]. This way an aggregation (aggregate) A is reified by a known kind of software entity, an object r, through which the program can treat A as a unit: r identifies the aggregation $A = Aggr[r]$ and provides aggregation operations on A. Such an r is called **aggregate object**, and its class aggregate (class):

> "**aggregate [class]**: A class that represents the whole in an aggregation (whole-part) relationship." [OMG00:UML, glossary]

The *extent* $ext(A)$ of an aggregation A is the set of the aggregated implementation instances. For instance, consider a list A of objects a, b, c and d: $ext(A) = \{a, b, c, d\}$ (figure 8). List operations like inserting and removing elements are provided by the list's aggregate object (indicated by UML's "lollipop"-symbol). To form a *concrete* list, the elements must be "linkable," i.e., have a reference to another element.

[15]From a different point of view, aggregation A_3 could as well be called a single-linked list of three trees.

One possibility is to let the first list element a, i.e., a *part* of the aggregation, additionally play the rôle of the *whole*, i.e., $A = Aggr[a]$ (figure 8a). (Out of uniformity, *each* element is aggregate object of the sublist starting with itself.)

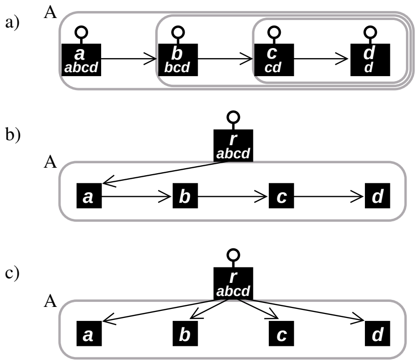


Figure 8: Aggregation with aggregate objects

To provide for an identification and for operations on a list that do not vanish once the list becomes "empty," a new *external aggregate object* r can be introduced into the system as a placeholder: $A = Aggr[r]$ (8b). The non-part r reifies that of the whole not found in any of its parts, that what makes the whole more than the sum of its parts.

How does one model a list of objects that have no interconnections among themselves, i.e., do not form a concrete aggregation? In this case, an *abstract* aggregation can be induced on them through an external aggregate object with a reference to each element (figure 8c). For example, our `Set` objects are external aggregate objects of *unbounded* abstract aggregations of arbitrary objects. Note that the elements constituting a set are not constituents of the set's external aggregate object of type `Set`, but represented by it.

Abstract aggregation, however, is no help for aggregating the *unbounded* number of `Node` objects introduced to represent unbounded `Set` objects by implementation instances. In order to establish a *concrete* aggregate, a `Node`'s state shall also comprise a model variable nx for a reference to another node. Then cyclic paths of nx references aggregate node instances to single-linked rings (middle of figure 9). This aggregation induces an abstract aggregation of the elements stored in the nodes. A `SetImpl` instance r together which such a ring can be the representation of a set object. All that is needed is to aggregate the two together by an anchor reference from r into the ring.

Abstracting. In §4.2, new data entities were introduced into the system besides the language's values by abstraction from compound values. Analogously, new objects besides those instantiated from class modules can only be added to the system by another act of abstraction. One or more implementation class instances together as an aggregation A can be abstracted from to obtain a higher-level, abstract object o. This o can then be the wanted aggregate object for the `Set` aggregation $A' = Aggr[o]$ (right side of figure 9). An aggregation A which represents an abstract object o is called *composition* or composite (object) in object-oriented design:

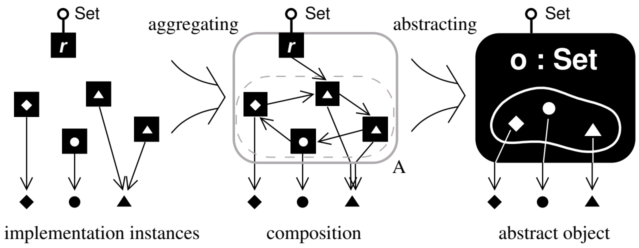


Figure 9: From instances to abstract object

> "A composite object is an aggregation that can be viewed from two levels, either in detail or as a single abstract object ..." [Rum94:ComBB]

> "A composite object represents a high-level object made of tightly-bound parts." [OMG00:UML]

The entire composition A together "represents" the abstract object o in the sense of data representation or model reification: The behavior of the abstract object (the whole) is realized through the combined behavior of the objects $Rep(o) = ext(rep(o))$ in its representation (the parts). "A composition may be thought of as a collaboration" [OMG00:UML] of all the parts, who are "tightly-bound" together to work as a *system* (runtime instance) [BLM97:HierEnc]. This system of parts simulates the abstract object; the relationship between the whole and the parts is a *functional* one [Civ93:ComAD].[16]

Whereas an object can surely belong to several overlapping *aggregations* at the same time, some techniques like UML exclude simultaneous membership in several *compositions*.[17] In practice, through, objects with no mutable state ("value objects") are naturally shared between representations: In design patterns, Context objects may share stateless *State* or *Strategy* objects [Ga+95:DP]; and in delegation-based systems objects share stateless "trait parent" objects (see §5.3).

Representative. One instance r in o's representation has the responsibility of implementing o's interface. Since o as an abstract object is defined by its interface, this r "represents" o vis-à-vis the clients, it is o's **representative**. The representative of an abstract object

[16] The relationship between an abstract object and model parts of its specification (§5.1) is also a functional whole-part relationship [Civ93:ComAD]. But it is one obtained by analysis of the abstract object, not by design, where composition's main use is [Rum94:ComBB].

[17] UML composition furthermore implies propagation of destruction ("life-time dependency") and bi-directional navigability.

o corresponds to the *dominant object* of the composition $rep(o)$ [Rum94:ComBB], and to the *root instance* of $rep(o)$ as a system [BLM97:HierEnc].

How do operation calls which clients direct to abstract object o arrive at implementation instance r? Both simply have the same object identifier $\iota \in OID$ which, depending on which level is looked at, means abstract object $o = AOb[\iota]$ or implementation instance $r = Inst[\iota]$. Therefore the same *dynamic binding (message dispatch) mechanism*[18] of object-oriented languages for handling calls of an implementation instance's operations handles, from a higher perspective, also calls of operations of the abstract object which that instance represents.

The representative's important rôle towards clients implies also an pivotal rôle within the composition. A composition has no need for an aggregate object to represent the whole aspect and provide operations pertaining to it *as an aggregation*, like enchaining and dechaining nodes. Instead, the representative r of abstract object o provides operations pertaining to whatever the abstract object models, and takes on the rôle of an internal aggregate object w.r.t. identifying the composition $rep(o) = Aggr[r]$. The implementation instances implementing o are the extent of $rep(o)$, and r is always one of them: $Rep(o) = ext(rep(o)) = ext(Aggr[r]) \supseteq \{r\}$.

Implementation. The class module is a construct specifically for defining in the program how an implementation instance is represented by a record at the language level (level 2). But no construct in the programming language allows the programmer to state how he designed the decomposition of abstract objects into implementation instances. This design information is mostly lost when the program design is implemented in the programming language. Recovering the (de)composition structure from the program code without the design information is not trivial:

In object systems, a reference from object a to object b is not only a means for aggregating a and b. It is also needed for communication: a invokes an operation of b "through" a reference from a to b. In order to turn a set of objects into a system and achieve computation in-the-large, these communication links must connect virtually all objects in the system. In the *Observer* pattern [Ga$^+$95:DP], for instance, a Subject has references to registered Observers to notify them whenever it changes. The "significant" distinction between references which glue objects together to aggregations or those who do not is a semantic one, "imposed" through designer's intent, but "hard to see" in the code [Ga$^+$95:DP].

Instead of using low-level semantics and representations' extent to infer high-level semantics, as in §4.2, one can use low-level semantics and a specification of the high-level behavior to reverse engineer the extent of representations. As opposed to ordinary aggregation, membership in compositions does not depend on the mere *existence* of "glue references." These references must also be *used* for the communication ("glue communication") which constitutes the computation "in-the-medium" that realizes the abstract object's functionality. And the patterns of this communication are encoded in the program ("glue code")!

The analysis of what belongs to the composition representing an abstract object o has to start at the object r implementing o's operations, i.e., at the representative. After all, these operations, and nothing else, define o as an abstract object (§5.1). r can handle each invocation of o's operations itself, or delegate (part of) the job to other objects, which may further delegate sub-jobs to others. All objects which contribute this way to realizing o's operations are part of the system simulating o, i.e., constitute the composition which is o's representation. These parts of the representation besides representative r shall be called r's **assistants**. Instances which are never reached by this recursive delegation play no rôle in realizing o; they are effectively not parts of the system (composition) representing o.

Class modules still only implement their own instances. But it can be expressed in many languages that instances of class `SetImpl` are representatives of abstract objects of type `Set`. In Java, one has to write `class SetImpl implements Set` as of class module's head. This declaration insures that `SetImpl` instances provide an implementation for each operation in interface `Set` ("interface inheritance"). The same declaration in a specification of `SetImpl` states that the behavior of `SetImpl` operations satisfies the behavior specified for the corresponding `Set` operations ("specification inheritance").

For our discussion it will suffice to look at the *specification* of `SetImpl` in figure 10, which abstracts from details of `SetImpl`'s implementation in appendix A. The nodes (the assistants) are characterized by a simple class-style data abstraction `Node` (c.f. appendix A) with the above mentioned model variables el and nx. The representative's class contributes in three major ways to implementing the abstract object:

First, a class, as a data abstraction implementation (§4.2), defines the concrete representation of representative r, i.e., of one implementation instance in o's representation. `SetImpl` is not interesting as a data abstraction. The state of its instances is completely modeled as a reference anc to a `Node` instance (the anchor into the ring of nodes). This model variable cannot be observed from the outside, since `SetImpl` implements only set operations. The specification of behavior w.r.t. to `Set`'s model variable has is inherited unchanged from

[18] To handle calls of an operation f of an implementation instance r identified by ι, the implementation of f is selected which is "most recent" in the order of r's class and its base-classes.

```
communicator & data abstraction
class SetImpl implements Set {
  spec anc : Node;   — implemented by private variable
  invariant anc ≠ null ⇒ ∀i• anc.nx.nx^i ≠ null
                        ∧ ∃i• anc.nx.nx^i = anc;
  public SetImpl()   ensures ∀Object o• ¬has(o);
  public boolean has(Object x)   observes anc.nx*;
  public void Insert(Object x)   mutates anc.nx*;
  public void Remove(Object x)   mutates anc.nx*;
}
```

Figure 10: Specification of class `SetImpl`

`Set`'s specification (§5.1).

Communicator abstraction. Second, a class defines for its instances a certain pattern of communication with other objects. This communication either concretizes the abstract object's communication with those objects or else is part of implementing the abstract object's data aspect by collaboration with assistant objects in the composition. A `Set` object does not communicate with its environment. But the `SetImpl` operations, on the other hand, observe and mutate other objects during their execution: Whether representative r tests for membership, inserts or removes elements, it will scan through the ring by starting from node anc, and by following nx links. This set of objects reachable from anc along nx links can be written as $anc.nx^*$.

For the representation's extent most details of the communication are irrelevant. It suffices to know for each operation which objects may be observed, and which may be mutated and observed: A mutates clause declares the objects whose mutators might be called (indirectly) during its execution. An observes clause declares the objects whose observers might be called. Objects named in the mutates clause can also be observed, without explicit naming in the observes clause. observes `this` is implicit for all accessor operations. mutates `this` is implicitly for all mutator operations.

Operation `has` is implemented by observing objects $anc.nx^*$. Thus, the model variable has : `Object` $\rightarrow$ `boolean` of a `Set` object o with a `SetImpl` representative r depends on r *and* on r's $anc.nx^*$ nodes, i.e., their `Set`:(`SetImpl` $\times$ $2^{\texttt{Node}}$) interpretation is o's entire state $\sigma(o)$. This tells us that objects $r.anc.nx^*$ are r's assistants $assi(r)$. Together with r they are the implementation instances $Rep(o)$ of o's representation.

$$\begin{aligned} \sigma(o) &= int(\langle\, \sigma(r),\ \{\sigma(n) \mid n \in assi(r)\}\ \rangle) \\ assi(r) &= \{r.anc.nx^i \mid r.anc \neq \texttt{null} \wedge i = 0, \dots\} \\ Rep(o) &= ext(Aggr[r]) = \{r\} \cup assi(r) \end{aligned}$$

In the general case, assistants can be abstract objects themselves (c.f. §5.3). Then $Rep(o)$ is the *transitive* reflexive closure $assi^*$ of $assi$ applied to r.

A communicating object may have certain condition w.r.t. its environment to function properly. A `SetImpl` instance, e.g., expects that its assistant nodes are closed to a ring so that there is no node without a next node. Instead of writing this as pre- and postconditions for all operations, the ring structure can be declared as a system invariant over the environments of `SetImpl` instances. In this case the invariant is `SetImpl`'s **level-3 representation invariant** since it talks about the objects in the abstract object's representation.

The specification does not explain (although, in principle, it could) *how* the assistants are changed, nor *how* observing them affect the behavior. Does `Insert` add also a node if the inserted element is already in the set? Does `Remove` dechain nodes or reset their element pointers? — These are design decisions hidden in class module `SetImpl`. However, these "details" could be specified in cases where subclasses, iterators or other objects are supposed to access the nodes themselves.

Constructors. The third and final contribution of the representative's class to the abstract object's implementation is to decide which objects become r's assistants: r can create its assistants itself, or obtain them from other assistants or from outside (c.f. *black-box reuse* at the end of §5.3). User-defined constructors in class modules were introduced to set up instance variables representing a new implementation instance (§4.2). In object-oriented systems, constructors of representatives can be and are programmed so that they also set up the abstract object's representation by installing appropriate assistants (c.f. the libraries of C++, Java, Smalltalk [GNU99:g++lib, Sun00:JavaAPI, GR83:ST]). This makes the construction of abstract objects and their representatives inseparable. — The constructors in the representative's class are actually constructors of the entire abstract object!

A special case of construction is cloning, i.e., creating a new object as copy of an old object. Cloning an abstract object (e.g., a *Prototype* [Ga+95:DP]) means a **deep copy** of the entire composition [OMG00:UML]: replicate all the (unshareable) implementation instances with corresponding references among them, and copy the references to objects outside of the representation. Language supported cloning as in C++, Eiffel, Java and Smalltalk is shallow by default, i.e., clones the representative (implementation instance) only. But the programmer can override this behavior in the representative's class to implement deep cloning himself.

To sum up, the class module not only defines the same representation structure for a set of implementation instances (the class's instances) but also for a set of abstract objects (whose representatives are the class's instances). The design decisions encapsulated in a class module (which a subclass reuses by inheritance) are not only about the instances' record representation and the operations' algorithms, but also about who the assistants are, and about how to install them.

5.3 Object-oriented and Component-based Software

In object-oriented systems implementation instances hierarchically combine to more and more complex abstract objects. This is consequence of "essentially all" object-oriented design methods being *compositional* [CLF92:POOD]: "Objects obtain their static and dynamic properties by composing, delegating, inheriting, and coordinating those of other objects. The basic idea ... is to build complex objects and operations out of simpler ones."

> "By incorporating state and behaviour, the concept of object ... allows us to regard each design step as the implementation of some abstract object in terms of a collection of concrete ones that are "assembled" into a configuration that provides the functionality required by the abstract object." [FM90:DSIO]

For example, a dictionary object o associates key values like student names with item values like grades (figure 11). o can have a representative which uses a pair object to hold each key:item combination, and a `Set` object as aggregate object of the set of all these pairs.

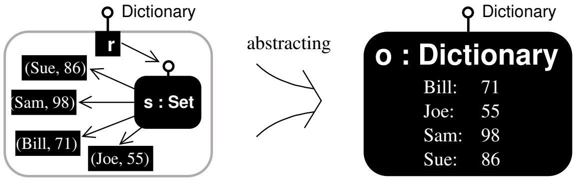


Figure 11: Dictionaries implemented with sets

Classical **object-oriented design**, among others, refines (decomposes) objects still deemed primitive in analysis [CLF92:POOD] recursively to objects at lower levels of abstraction [Boo94:OOAD]: A design object o is decomposed into a representative r, and assistant objects n_i, which are decomposed further. Even if o could be represented without assistants it may be decomposed further to take advantage of reuse opportunities or to obtain more manageable or reusable pieces. Decomposition ends with representatives without assistants.

Inheritance (§4.5) can be seen as a language supported static definition of composition with special semantics. Here an assistant "sub-object" corresponding to the instance defined through the base-class is combined with the representative "sub-object" corresponding to the "delta" defined in the subclass. This is more apparent in **delegation-based systems** like Self [US87:Self] which have no classes but support inheritance between objects by the delegation mechanism. "[A]n extended object is implemented as if having 'components' " [Ast96:ReoSpl]: the representative, "child," with the new features, and the hierarchies of unshared assistants, "data parents," to hold the object's inherited state, and of shared assistants, "trait parents," which supply the (inherited) methods.

Component-based software rests on the notion of *software components*. They are an abstract object's (compiled) implementation, i.e., the packaging of the collection of (compiled) class modules which implement all the implementation instances into which the object is decomposed. The packaging of collections of class modules to software components is not bound to a language construct but defined in the development environment. Tools for the reuse and new development of components are traditionally most successful with components that can be represented and manipulated *visually* (c.f. JavaBeans [Ham97:JB]). "Considerable examples of small components include many of the available GUI components" [Hop00:Com].

Component-based software development based on black-box reuse (c.f. further below) constructs the same hierarchical structure bottom-up which classical design constructs top-down: Abstract objects n_i already implemented by software components are assembled (composed) hierarchically to a more complex abstract object o with a new implementation instance r as representative. Correspondingly, the n_i's implementations are composed to the software component implementing o with r's implementation as glue code.

On the class module level, libraries or the market have to be searched both for an implementation of a representative (e.g., class `SetImpl`), and implementations of each type of assistant (here class `Node`). This allows mixing different implementations of representatives with different implementations of assistants, and creating independent objects and assistants for other representations from the implementation of one representative's assistants. But "piecing together a complex system from a catalog of parts won't work for anything beyond the simplest applications" [Cha97:CSEM].

In component-based software, the objects' decomposition into parts parallels the partitioning of system design/implementation into subsystems. A subsystem is the complete specification/implementation of an object, its instances are "subsystem objects" [Rum94:ComBB]. Software components, the packaged code of subsystems, supersede class modules as the unit of reuse, distribution and versioning—just like class modules (i.e., collections of procedures implementing a data abstraction) superseded single functions (§4.3).

> "[C]omponents are generally much easier to use than class libraries. Using a class library often requires understanding how the various pieces of the library fit together, something that can be a complex undertaking for powerful class libraries." [Cha97:CSEM]

Abstract objects, not single implementation instances, are units large enough to be component objects with a bit-level interface in component architectures (§5.1). Abstract objects also are a good granularity for concurrency, locking, clustering, persistence, etc. [Rum94:ComBB, Ki$^+$87:ComDB].

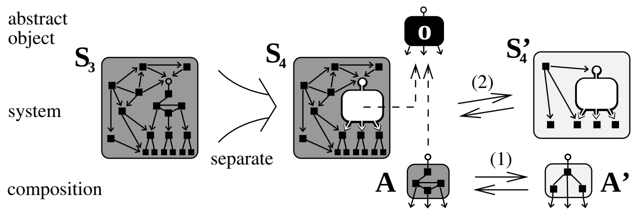


Figure 12: Component-based system architecture

The use of abstraction. Normally, clients of objects "are interested only in an object's behavior" promised by its "contract", and and "not [in] the implementation it uses to fulfill that contract" [MD95:COM]. This is codified as the first principle of object-oriented design in [Ga$^+$95:DP]: "Program to an interface, not an implementation." This means that clients use an object identifier ι as the abstract object $AOb[\iota]$ as characterized by its interface and specification, not as the corresponding implementation instance $Inst[\iota]$. An analogous design rule is that what clients should declare about object variables is the object type, not class [Sou97:TaC].

The representative is also just a client of its assistants, using them for its own purpose [MD95:COM]. Hence, the representative's unit of assistance is the abstract object (c.f. *black-box reuse* further below). The only a client must name a class is to instantiate it." The creator of an abstract object knows its implementation since it has to instantiate the representative's class (§5.2). Hence creation and use of an object are often separated, as explained further below (*flexible object creation*).

Let S_3 be a system of implementation instances comprising a set A of objects. If A can be identified as the representation of an abstract object o then it can be separated from the subsystem S_4, which is client of o (figure 12). The explicit specification of o's characteristic properties (c.f. the `Set` specification in §5.1) can be seen as a contract similar to that between modules in §4.3: Any composition A which is supposed to be a representation of o has to realize these properties. And only these properties a well-designed client system S_4 may rely on, and it must not interfere in the representation's inner working. Then it is possible to safely reuse of S_4 and of A in different contexts:

1. *Object substitutivity*: The choice of an actual representation A for object o should have no effect on clients in S_4—even if the client is a representative and o is its assistant. Then one representation A of o can be replaced by a representation A' of o in the same state without that clients can notice it. Then the system S_4 working with the first representation of o can be reused unchanged with arbitrary other representations of o. For example, a set o represented with a ring of nodes A could be replaced by a faster representation A' with an internal red-black tree, as in gcc's implementation of C++'s `set` class template [GNU99:g++lib]. It should even be possible for the new representation A' to implement a subtype of the object type which A implemented [Lis88:DAH].
2. *Object integrity*: The representative r can establish and maintain invariants and an access control regime over the entire representation A. Hence what represents abstract object o in one system S_4 can be reused safely without adaptions in any other system S_4' which needs a working object o.

Object substitutivity and integrity are about reuse in execution environments where an object, i.e., a run-time entity, can be substituted for an old one in a running system, or transfered from one into another running system. This is crucial for evolving open (distributed) systems like those of the CORBA architecture. Here, a component object with arbitrary representation can enter the system anytime from anywhere in the network: It establishes communication with the system's "object broker" and registers as server or requests to be linked with a server object.

Runtime substitution is possible where object implementations are identifiable packaged subsystems [Rum94:ComBB], i.e., software components. The reuseability of an object's implementation in many systems is ensured by the integrity of the object. Parameterization is a language mechanism for defining statically a substitution of values or objects at runtime: Operations with a formal parameters of type `Set` will, due to substitutivity, behave the same whether called with sets represented by rings of nodes, or by red-black trees. More substitution can be achieved statically if object creation is organized in a flexible way:

For **flexible object creation**, the system part F deciding about the creation of an abstract object o has to be separated from the system part C only using it. Even assistants' creation can be separated of their use by representatives: Either F calls a *configuration operation* belonging to C to pass different compositions A_i as parameter which C should use as the object o from now on. Or creation is encapsulated in a *factory operation* F which C calls to obtain a composition A as result and to use as the object o from now on. The latter is canonized as the design patterns *Factory Method* ("pervad[ing] toolkits and frameworks" [Ga$^+$95:DP]), *Abstract Factory* (fundamental in COM), and *Prototype* (the basis of the prototype-based programming). C can be combined with a different representation A' of object o by switching to an implementation of F which instantiates representatives from a different class. "Switching" in turn can be achieved by revising the class module where F is implemented, or by substituting the object whose F operation C calls to one with a different implementation of F.

External determination of assistants turns an object's implementation from a pre-determined set of classes into an open mini-framework. For instance, a scanner object o can have a model part *source* modeling the stream of characters tokenized by o, and an operation for setting the reification of *source* [DLN98:Wrest] (a *Strategy* object [Ga+95:DP]).

Black-box reuse is the use of objects with old implementations in new compositions. For producing assistant implementations like `SetImpl`'s class `Node` that are reusable, assistants should be designed free of the context of a specific representation [VMO99:ESCO]. Black-box reuse, based on object reusability, is an alternative to module reusability-based class inheritance (white-box reuse) [Ga+95:DP, MD95:COM, Pre97:KSF], with advantages in four areas:

(1) Subtyping allows reusing client code with objects of different implementations. Inheritance allows reusing an object's implementation. Both are, in principle, independent [Sny86:EncInh, Ga+95:DP, Bru96:TyOOL], so that separating them should increase the potential for reuse. But programmers seem to have difficulties in working with separate subtyping and inheritance hierarchies at the same time. Separate is-a (subtyping) *and* has-a (aggregation) hierarchies, however, have a long tradition. Nobody is confused that operations useful on node objects make no sense as operations on the set object which uses nodes in its representation. Thus black-box reuse should make it easier to capitalize on the principal independence of reusing clients and reusing implementation.

(2) Inheritance fixes statically from how many and from which base-classes a subclass inherits. The life-time of the "subobjects" defined by each base-class module coincides with the entire object's life-time. Composition allows deciding at runtime from which class to create an assistant, switching between assistants (e.g., to reflect state transitions between different behavior modes [Un+91:PWC, Ga+95:DP]), and creating additional assistants at any time. These assistants may also be determined externally, allowing for dynamic (re)configuration (c.f. *flexible object creation*), and can exist before the abstract object's creation: In UML, old assistants can be explicitly removed from one composition and added to another, or remain independent, and thus survive the abstract object's death [OMG00:UML]. And the COM architecture allows an "outer" object to dynamically incorporate externally created "inner" objects [MD95:COM].[19]

(3) Method overriding raises the instable base-class problem because of undocumented self-calls between operations of the same object (§4.5). Composition makes overriding more manageable (and flexible). First, it separates overriding w.r.t. the clients' calls from overriding w.r.t. self-calls: The original object o is wrapped as assistant into a new object whose representative simply forwards to o the clients' calls of non-overridden operations only. o's self-calls can be redirected by subclassing the class of o's representative. Alternatively, o could offer configuration operations for setting the assistant who defines its responses to a certain call.

Second, replacing o's assistants actually corresponds less to overriding a single operation, but rather to replacing an entire "class component" [Sta97:MSC] at once, i.e., a meaningful subset of methods with related instance variables. Since assistants are objects, they have an interface that is well understood and can be clearly defined [MD95:COM, Ga+95:DP], so that they can be replaced safely. This contrasts to the poorly understood interfaces between sub- and base-classes.

(4) A subclass cannot override a superclass's instance variables, so that the representation of subclass instances can never be smaller than that of superclass instances. Instance variables can only be overridden *together* with the operations accessing them. This can be supported by a language extension like *class components*, an encapsulated group of instance variables and operations [Sta97:MSC].

With object composition, no special language construct is necessary: Class components are reified as assistants, created through Factory methods [Ga+95:DP] so that subclasses can override which classes implementing the assistants.

5.4 Object Encapsulation

The wonderful world of abstract objects and components falls apart if their abstraction boundaries are not preserved. Frontal and lateral breaches of the object abstraction are related to the (mis)use of references to, respectively, the representative and its assistants.

Frontal breach. References are occurences of an object identifier $\iota \in OID$. References to the representative $r = Inst[\iota]$ are unavoidable since are needed to identify, i.e., to reference, and use the abstract object $o = AOb[\iota]$. The `Set` abstraction boundary around an object o with a `SetImpl` representative r is breached frontally if clients dereference object identifier ι to something else than an object of type `Set` or a supertype. Misinterpretation as an object of the wrong type is easily excluded by another extension of the system of kept-apart types:

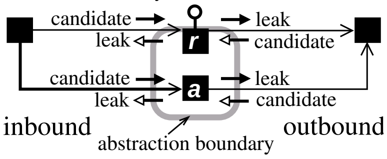
1. the types of the language's values (§3)
2. types of data with a user-defined representation hidden in an implementation module (§4.4)
3. and now, types of abstract objects with a representation only known to their own representative.

[19] Objects cannot be incorporated any more after having been asked for their so-called IUnknown interface [SM97:ThCOM].

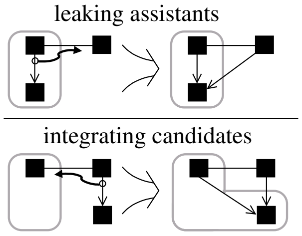
Object types are defined separately from their implementation, either by a special construct like `interface` in Java or by a conventionalized use of the class construct as a *purely abstract class*. Interpreting ι as an instance $Inst[\iota]$ of an implementation class is also a breach of the abstraction of object $o = AOb[\iota]$, since it means singling out the representative from o's representation. This should be excluded by not allowing type-casting from abstract object types to implementation class types, or by not using classes for typing purposes at all (as, e.g., in PolyTOIL [Bru96:TyOOL]).

Lateral breach. The exterior of an abstract object o are all objects and system parts not belonging to o's representation. o's interior is o's representation except for the representative, i.e., (the representation of) the assistants. Direct access of the exterior to the interior is only possible by means of an *inbound reference* or "exposing alias," i.e., a reference from the exterior to the interior (some assistant a). A *lateral breach* of the abstraction of an object o is a misuse of an inbound reference, i.e., a use by the exterior that is not part of (the realization of) o's behavior.

A much more detailed discussion than at the other levels will be necessary, since there is no simple cure against inbound references: One cannot simply have no object references at all—they are indispensable for computation in-the-large and for aggregating a representation in the first place (c.f. §5.2). And the *inbound* ones can be extremely useful to the implementation of objects. Inbound references originate from the passing of references across the abstraction boundary [DLN98:Wrest, NVP98:FAP]:



Either an object in the representation (representative or assistant) *leaks* a reference to assistant objects to the exterior (as result or parameter). Or the exterior retains a reference to an object, a *candidate* assistant, which it passed to the representation (as result or parameter) where it is integrated as a new assistant.[20]



Which inbound reference to assistant a has nothing to do with o's behavior is not always so easy to say, since it may be how the representation of polylithic object o implements the reification of o's model part α. Exposure and assistant in such a case shall be called *public*, in analogy to public instance variables, the client-accessible representations of model variables (§4.5). The opposite, *private* exposure, has nothing to do with the object per se, but *only* with its implementation. That is, it is an implementation detail like private instance variables.

[20]Additionally, instances of Java's inner classes can "steal" references to the enclosing instance's assistants.

This gives eight cases of passing across abstraction boundaries, which will be considered in the following.

assistant	candidate		leaked	
	as param.	as result	as param.	as result
public	`SetCorner`	`mkCorner`	`Visit`	`corner`
private	`AddAll`	`getIter`	`NodeIter`	`contents`

Public boundary crossing shall be explained with the `Polygon` example (c.f. §5.1). Remember that the state of `Polygon` object o comprises `Pt` model parts $o.\alpha_{1,\ldots,n}$ for the corner points, which can be set and got by operations `SetCorner` and `corner`. With indirection through another object e, passing through o's exported interface can be replaced by passing through the interface imported from another object x, reversing the rôle of parameters and results. `Polygons` could offer a *Visitor* service [Ga+95:DP] allowing clients to name an object f, a visitor, for gathering information about the model parts (observing visitor), or for making mutations to the model parts, like moving each corner 10 units to the right (mutating visitor). This end, o's operation `Visit` takes visitor f with operation `Accept`, and, for each corner $o.\alpha_i$, calls f`.Visit(`a_i`)`, where a_i is some reification of $o.\alpha_i$. Finally, `Polygons` may allow clients to name an object f through some operation `ConfigCornerFactory` as the "default corner factory object." When the number $o.n$ of corners in `Polygon` o is increased by operation `SetN` from n to n', o will use the results of calls f`.mkCorner(`i`)` for $i = n+1, \ldots, n'$ as the new, not previously existing corners $\alpha_{i=n+1,\ldots,n'}$.

A representation A of o may use assistants $a_{1,\ldots,n}$ of type `Pt` such that each assistant a_i reifies a model part α_i. Representation A could implement the request `SetCorner(`k,x`)` to set o's corner α_k by making the given x the new assistant a_k which reifies α_k. The request `corner(`k`)` for o's corner α_k could be implemented by returning a reference to assistant a_k. A could also supply references to its corresponding assistants to the visitor as parameters in the `Visit` operation. And finally, A could simply integrate the `Pt` objects returned by the `mkCorner` operation of the "default corner factory" as new assistants for representing the new corners.

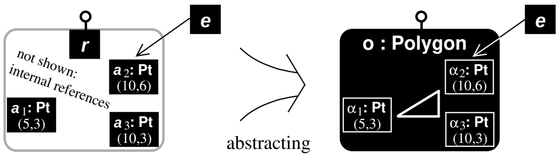


Figure 13: Public exposure

Either way, in the end the exterior may come to hold a leaked or retained reference to an object a which is an assistant in o's representation (see figure 13). But

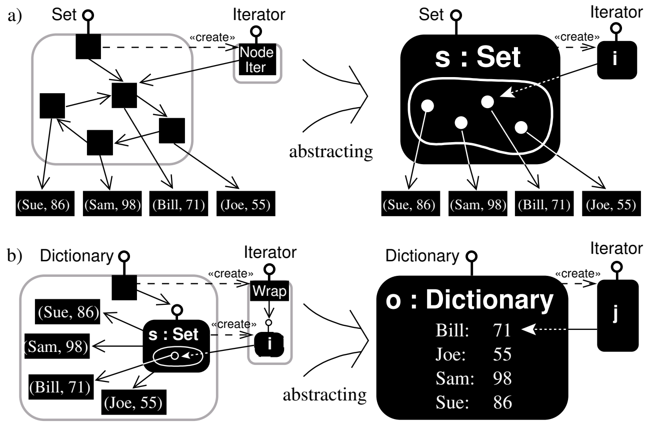


Figure 14: Iterators for sets and dictionaries

this a is the reification of a model part α of `Polygon` o, and the exterior knows this, since it is part of o's behavior. Hence the public exposing reference into the representation can be understood as a reference to a model part of the abstraction o. And the exterior knows from o's specification how α and thus its refication a relate to o.

An alternative representation A' might work without `Pt` assistants. Using "transporter exchange" (§5.1), A' would implement `SetCorner` and `SetN` by reading the coordinates from the obtained `Pt` object x and storing them at the appropriate place. `corner` and the `Visit` calls in `Accept` could be implemented by creating a new `Pt` x into which it stores the current coordinates of o's corresponding corner before passing x as result or parameter, respectively.

Private boundary passing has more diverse uses, as shall be explained by means of `Set` o and its representation A with a ring of nodes.

Parameter leak detail. Objects may chose to implement their own behavior by using the service of an external object e to operate on their representation. To this end, a reference to internal objects is leaked to e, typically in form of a parameter. This is analogous to using a `swap` procedure for swapping instance variables (c.f. §4.4). For example, there may be objects in the system providing a service, *friendly function*, for removing duplicate values from a node ring, for cloning a node ring, for comparing or merging node rings, and so on. Representation A may use these objects to ensure that each element of o is stored only once, to implement the cloning of o, set comparison, set union, and so on.

Another case is that where the abstract object o has an operation to return an object x whose behavior depends on aspects of o's state that no object can observe through o's other operations. For example, set o could provides *iterator* objects [Ga$^+$95:DP] to its clients through operation `getIter` (figure 14a). Each returned iterator independently enumerates the elements in the set: one operation returns a current element and another one advances to the next. This allows keeping the `Set` interface small. An iterator is an object of its own, with its own state (the state of the iteration over the set).

If no operation in o's interface allows to obtain the information which an iterator i needs, and if we assume that the set's contents was not copied into i, then i must extract this information itself from representation A, and thus (a part of) o's representation has to be exposed. This is the essence of the *Iterator* design pattern: the iterator traverses directly through the set representation. This requires the iterator to be compatible with the representation, in this case a `NodeIter`, and hence it is normally created by the set representation itself. The necessary inbound reference is typically passed to them as a constructor parameter when they are created by the set.

Result integration detail. There can also be iterators for the entries in a dictionary. In case of a dictionary object o with a set assistant s storing the entries (§5.3), iteration over o's entries means iteration over s's elements (figure 14b). Hence, o's iterator j has to integrate as assistant the iterator i which it obtained as result from s`.getIter()`.

Parameter integration and result leak detail. `Set` objects might offer a service for inserting an arbitrary number of objects in one step, and a service for returning all its elements in one step. In order to pass all these objects through the `Set` interface at once, they are aggregated in a `Node` list L, and the reference to L's head node is passed, respectively, as parameter to the operation `AddAll` for insertion, or as result of the operation `contents`. The representation A of `Set` o with a ring R of nodes could implement `AddAll` by simply integrating the given list into R. And it would implement `contents` by returning L as a new list, while another representation A' with a node *list* L' could simply return L' as its `contents`.

A new object representation for o can stop old private leakages and integrations and start new ones, so that private exposure can follow any change in the representation: An alternative representation A' of sets with an array or a balanced tree instead of a ring will simply use another kind of iterator and other friendly function objects. And an alternative implementation of the dictionary's iterator might not integrate the set's iterator as assistant (but use it immediately to obtain and then remember all entries), or might not call `getIter` at all but use `contents` to obtain all entries in a list.

Substitutivity. Object types are the contracts specifying the properties of an object o which its representations must satisfy and thus which to expect is justified for o's clients (the exterior). If clients have unjustified expectations then there might be representations not satisfying them, so that substitutivity of representations for o is lost in the context of such clients. This reduction of reuseablity of the client part S_4 of the system is its own fault, since it was the one who violated the contract.

Clients using o's operations, and analogously, the representation implementing o's operations might not expect the visible consequences of exposure (or its absence, depending on o's representation): Callers of `Set` o's `AddAll` and `contents` operations, and maybe also of `Polygon` o's `SetCorner` and `corner` operations, may be surprised to find that through the object x which they exchanged with o they sometimes observe o and cause mutations to it, and at other times they do not. It might come as a surprise to objects returning another object x as result of operations called `mkCorner` or `getIter`, that after returning x, x may co-evolve with some other object o (because it integrated x). And, in the case of the `Visit` and `NodeIter` operations, any object might be surprised if the object x it obtained as parameter co-evolve with some other object o.

But the real source of the problem will often be that the object type actually did not specify how x and o will co-evolve after the call. This may easily happen, because co-evolution is difficult to express in detail. The only behavior simple to specify is identity exchange of model part reifications (§5.1), which implies a co-evolution guarantee (until the model part is given a new reification). It is however no satisfying solution to simply say that specifications are too weak for justifying any expectation about (effects of) exposure. Especially, in cases where a parameter or result x is to be integrated as assistant it will be important to have specified that x does not already co-evolve with any object, i.e., is not internal to any object's representation, so that x can be used as assistant without surprises.

Integrity. Except for identify exchange of model part reifications (§5.1), it is always the representation's choice when to integrate and leak assistants. Lateral breaches of integrity can be avoided if the representative and the interior use references to the exterior for handing out internal references only when the exterior will use them in an integrity conforming way (temporary violations may be acceptable). In the case where o's representation A wants to obtain or leak assistants a by calling operation m in the imported interface of another object f, A has to ascertain this from m's specification in f's type. (It does not matter whether some client of o knows about this exposure, so that the exposure is not private.) Current specification techniques are good at describing how a is used during o's call of f's operation. But it is much more difficult to express, and thus unlike to be included in the specification of f's operations, how a is used *after* the call. If the specification is too weak then it is unsound to leak or integrate assistants which are constrained by o's model (history) invariant or by A's representation (history) invariant.

> **Security hole in JDK 1.1.1** [SIP97:SigBug]:
> For every Java class C there is an abstract object o_C in a running application to return runtime information about C (for reflection), like C's name, superclass, operations, and so on. Since properties of Java classes do not change at runtime, immutability of these pieces of information is a model history invariant of o_C:
> $name = \mathsf{old}\ name,\ super = \mathsf{old}\ super, \ldots$
> o_C`.getSigners()` returns C's digital signatures in an array. Browsers can be told to relax security restrictions for applets instantiated from classes with certain "trusted" signatures. The representative r_C of the o_C object, an instance of class `Class`, uses an array assistant s to hold C's signatures. Since a class's signatures never change, it is a level-3 representation history invariant for r_C that the elements in its s array never change:
> $\mathsf{size}\ s = \mathsf{size\ old}\ s \land \forall i = 0 \ldots \mathsf{size}\ s \bullet\ s[i] = \mathsf{old}\ s[i]$.
> In JDK 1.1.1, r_C implemented o_C's `getSigners()` operation by returning their assistant s_C. Everything is fine with this, as long as o_C's clients preserve r_C's representation invariant w.r.t. s_C. But malicious applets of class C may ignore o_C's abstraction boundary and violate its integrity by overwriting their class's signatures in s_C with other ones, until they find a trusted one. This way they can rid themselves of security restrictions.

The other case is that o's representation A wants to obtain or leak assistants a through operation m in the exported interface of an object ω in the representation which was called by the exterior. ω's type needs to specify to its clients the constraints on using the passed object a which are necessary to preserve o's model, and A's representation invariant. If the specification is too weak then it is unsound to leak or integrate assistants constrained by invariants. For example, leaking a private assistant without an associated constraint on its usage lead to a hole in Java security, as described in the box on page 24.

The only case where classical specifications, with operation properties and (history) invariants [MP01:Univ], help is public exposure through the exported interface: Whenever the exterior sets or obtains an object a as reification of model part α through the exported interface of object o then it must assume the worst—that a is (now) a public assistant (unless o's type explicitly specified the opposite.) From o's type the exterior also knows the model (history) invariants possibly constraining α. Consequently, in this form of public exposure, as with model invariants over public instance variables (c.f. §4.5), it is the shared responsibility of representation *and* exterior not to violate these model invariants (and additional representation invariants would be unsound). Then object integrity means that clients who violate a model invariant by mutating x have to reestablish it before this is noticed by anybody, in particular by the representative. Integrity in specification system Fresco [Wil92:FMOOP] is based on this: here *all* assistants are public and all invariants are model invariants. However, doing away with private assistants is no real solution either.

Exposure disciplines. In small software projects, the right intuition about the structure of objects may make careful programmers automatically avoid causing lateral breaches with inbound references. However, this is too unreliable for commercial projects, robust software components and open component architectures. Instead, the type system, the provision against frontal breaches, should be complemented by a discipline of how to use inbound references (exposure), in order to protect also against lateral breaches. Only then are objects encapsulated all around.

Object encapsulation should allow for "*modular representation reasoning*," i.e., reasoning about the object's representation, its state space, the representation and model (history) invariants, by looking only at the code (implementation class module) of the representative, and at specifications. Doing this with classical operation- and object-property based specifications requires an exposure discipline characterized by non-local properties which answer the following questions:

P1 How does the exterior use leaked references?

P3 How does the exterior use retained references?

P2 How does the interior and how does subclass code use references?

P4 What is internal and what is external? That is, where are the abstraction boundaries around representations?

The properties may or may not be declared explicitly in the program. Techniques for ascertaining properties of reference use (P1-P3) are called *alias control* as, for instance, presented at the International Workshop on Aliasing in Object-Oriented Systems [Nob99:OAl]. "Whereas alias control is a nice programming feature, it is a *necessity* for modular verification of object-oriented programs" [MP01:Univ].

In the literature, the most common answer to retained references (P3) is to avoid them all together by integrating only objects to which no other references point (no aliases). It is typically assumed that newly created objects, e.g., clones of given objects, are not aliased. Note, however, that user-defined constructors can well leak reference into the representation as parameters.

Based on the use of leaked references (P1), five disciplines can be distinguished:

(1) **Representation containment** [CPN98:OT] is a discipline which works without ascertaining properties of the exterior's use of inbound references (P1).

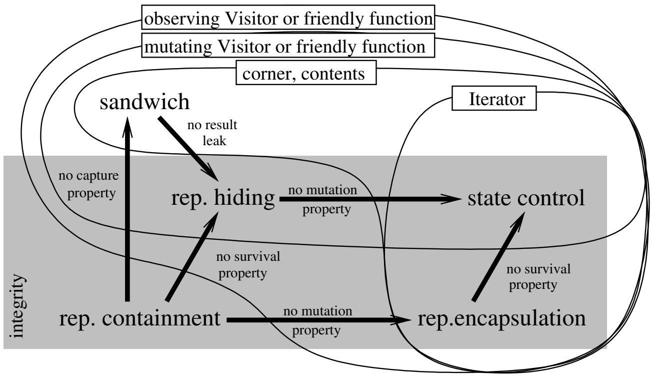


Figure 15: Exposure disciplines

It corresponds to the abandoning of pointers to variables and by-reference parameters for modular encapsulation (§4.4): There are no inbound references at all, so that all observations and mutations have to be made through the representative. Containment is the most common understanding of "object encapsulation," appearing in form of Wills's *type isolation* [Wil92:FMOOP], de Champeaux et.al.'s *ensembles* [CLF93:OOSD], Civello's *assemblies* [Civ93:ComAD], Almeida's *opaque balloons* [Alm97:Bal], Noble, Vitek and Potter's *flexible alias protection* [NVP98:FAP], Clark, Potter and Noble's *ownership types* [CPN98:OT], Genius et.al.'s *internal classes* [GTZ98:Sand], and Vauttier et.al.'s *internal composition* [VMO99:ESCO]. Also coding guidelines like leaking and integrating only clones [Eng01:Geh] lead to the containment of representations.

Since all access is routed through the representative, modular representation reasoning can be done based on only the specifications of assistants' operations called by the representative. The violation of the representation and model invariants by such a call can be predicted so that the representative can reestablish them before itself returning control to the exterior. Hence, from the outside, the invariants are, despite temporary violation, "data invariants," i.e., properties holding whenever "control is not in the object" [Set97:PLCC].[21]

Without inbound references, Iterators, Visitors, friendly functions, `corner` and other uses of leaking an assistant a cannot be implemented (in the described way). Instead, a has to be cloned and this clone has to be handed out. That the representative does this when passing objects through its exported interface or through imported interfaces can be ascertained by looking at its code. In Eiffel, for example, the compiler ensure that the representative does not leak references

[21]This extends to composed objects the temporary violation of invariants commonly accepted for implementation instances [DL97:SpecInh, Wil92:FMOOP]. Temporary violation however raises the problem of call-backs in that situation for which there is no commonly accepted solution.

to an assistant a declared to be "expanded." On the other hand, nothing prevents a from using a reference to an external object e to pass itself to e by a call like `e.TakeMe(this)`. But how can the representative know from the assistants' operation specifications that they do not use outbound references to an object e for leakage through e's imported interface (P2)?

One could try to control the assistants' use of outbound references from the representative's module, by not passing them any such references in the first place. Then they can only communicate with the exterior by relaying calls through the representative.[22] This allows, in principle, intercepting any passed assistant.

Opaque balloons, ownership types and the related flexible alias protection are techniques in which abstraction boundaries can be defined (P4), and which prohibit assistants and the representative from leaking internal references to the exterior (P2). In ownership types, only newly created may be added to the representation. The other two techniques provide for references with the property of having no aliases, and ensure that only such unaliased references are used to integrate a candidate assistant. This way all three techniques avoiding retained exposure (P3) and thus guarantee containment of the representation.

(2) In the **sandwich** discipline [GTZ98:Sand], inbound references may not be "captured," i.e., not stored in instances variables nor global variables. That is, the reference use property P1 is the reference's storage place. This simple property alone is too weak for modular representation reasoning (but it may be strengthened, as explained below): Inbound references can be leaked as results so that their use cannot be reasoned about any more based on the operational specification of the operations called by the representative. Sandwiches allow all cases of exposure discussed, except for Iterators, which capture an inbound reference in an instance variable.

Hogg's *islands* [Hog91:Isl], the related *assemblies with inward visibility* by Civello [Civ93:ComAD], and Almeida's *balloons* [Alm97:Bal] are sandwiches. Islands and balloons are techniques which ensure the no-capture property (P1), and the property of having no captureable aliases (P3). Based on a simple definition of abstraction boundaries (P4), they enforce that representative and assistants leak only uncaptureable references and that they integrate as assistants only new objects, or, in case of islands, only candidates pointed to by non-captureably-aliased references (P2). This way both techniques guarantee uncaptureability of leaked and retained inbound references.

(3) **Representation hiding** (a "concept of hiding" [Min96:AlFr]), corresponds to limiting pointers to variables to the form of by-reference parameters (c.f. §4.4): Inbound references are only used in calls (indirectly) made from the representative's operations. The reference use property P1 is temporal: how long can the reference be used (the reference's life-time), in particular can it "survive" an operation call?

Control in some sense aligns the communication (data flow) between representation and exterior via exposed mutable assistants with the procedural flow of control. This excludes Iterators and result leaking `corner` and `contents` operations. But it enables modular representation reasoning based on the specifications of the called operations of assistants *and* external objects. Violated invariants can be predicted and reestablished.

Sandwiches in which the representative does not leak uncaptureable references would be hidden. Hogg may have had this in mind when he wrote "the state of [a representation] remains unchanged between methods invoked on the [representative]" [Hog91:Isl]. However, modular reasoning cannot ensure this property based on the sandwich discipline, since operations in a subclass could still leak uncaptureable results. Balloons ensure that inbound references to assistants other than "balloons" are not leaked as results, so that balloon-less representations are completely hidden.

Minsky's *u-pointers* [Min96:AlFr], and Greenhouse and Boyland's *object-oriented effects system* [GB99:OOFX] are two techniques guaranteeing representation hiding: They enforce declarations of the no-survival property (P1), and of having no aliases at all (P3). And, based on a simple definition of abstraction boundaries (P4), they ensure that representative and assistants leak only non-surviving parameters, and integrate only candidates pointed to by non-aliased references (P2).

(4) **Representation encapsulation** [MP01:Univ] focuses on mutation: the exterior uses inbound references only for *observations*, not mutations. Since then all mutations must go through the representative this enables modular representation reasoning based, as in containment, only on the assistants' operation specification. But it also conservatively excludes all mutating Visitors and friendly functions.

Representation encapsulation requires making a distinction between observers and mutators and identifying references restricted to call observers only (read-only). This distinction is available in C++ in form of `const` [ISO98:C++], and can be simulated in Java by explicit "const interfaces" [Eng01:Geh]. The coding guideline is to hand out references to assistants only as `const`. This way the representation obligates the exterior to use exposure only for observations (P1). But from the representative's module it cannot be verified whether assistants leak only references declared as read-only (P2). Hence C++ *enables*, but does not support, representation encapsulation.

[22] This can only be guaranteed from within the representative's module in languages where an object cannot read references from static variables.

Müller and Poetzsch-Heffter's (object) *universes* [MP01:Univ] allow the declaration of abstraction boundaries (P4), enforce declarations of the read-only property (P1), and ensure that representative and assistants leak only read-only references (P2). and integrate only newly created objects into the representation (P3). Thus representation encapsulation is guaranteed. And Islands ensures that inbound references to objects other than "islands" are read-only, so that island-less objects are representation encapsulated.

(5) **State control** combines the ideas of representation encapsulation and hiding: inbound references are used for mutations only in calls (indirectly) made from the representative's operations. Hence the representative "is in control" of all state changes in the representation. This enables modular representation reasoning based, as in representation hiding, on the specifications of the called operations of assistants and external objects. And it is flexible enough not to exclude any of the discussed uses of exposure.

Kniesel and Theisen's *access control* [KT99:JAC] enforces declarations of read-only and no-survival properties. A representation could use such declarations to obligate the exterior to use exposure only for observations or only during the representative's calls (P1). But from the representative's module it cannot be verified whether assistants leak unrestricted references (P2). Hence access control *enables*, but does not support, state control (and likewise representation hiding and representation encapsulation).

Detlefs, Leino and Nelson's *no abstract aliasing* method [DLN98:Wrest] enforces the properties of no survival (P1) and of uncaptureability and "read-only by specification" of all aliases of a captured "virgin" candidate (P3). Based on a simple definition of abstraction boundaries (P4), the method ensures that representative and assistants leak only non-surviving parameters, and integrate only "virgin" candidates (P2). This way retained exposure is read-only and leaked exposure is controlled by the representative. However, the method is more strict than necessary for state control, excluding Iterators and assistant leaking `corner` and `contents` operations. Hence no abstract aliasing is more like representation hiding with the only addition that (read-only) references can be retained to integrated assistants passed as parameters, as in the `SetCorner` case.

Discussion. Unfortunately, the proposed nine techniques for enforcing object encapsulation protect only representations with a specific internal structure [Hog91:Isl, Min96:AlFr, Alm97:Bal, DLN98:Wrest, GB99:OOFX], protect only the reifications of model parts [Wil92:FMOOP, DLN98:Wrest], are not safe w.r.t. modular representation reasoning [Hog91:Isl, Alm97:Bal], do not allow for externally created assistants [Alm97:Bal, CPN98:OT, MP01:Univ], do not allow returning assistants as model part reifications [Min96:AlFr, NVP98:FAP, CPN98:OT, DLN98:Wrest, GB99:OOFX], or do not allow iterators [Hog91:Isl, Min96:AlFr, Alm97:Bal, NVP98:FAP, CPN98:OT, DLN98:Wrest, GB99:OOFX]. The latter limitations are not acceptable: Open systems and black-box reuse with all its advantages (§5.3) *build on* passing externally created assistant candidates into the representation as parameters of configuration operations, or as results of factory operations. Polylithic objects naturally leak assistants as model part reifications to avoid polluting their interface with transitive expansions of model part interfaces [BC87:PartHier, VMO99:ESCO, HLS00:ObInt]. And iterators are a ubiquitous design pattern [Ga+95:DP] for a clear, powerful and efficient way of working with collection objects, like `Set`.

Cloning is no acceptable general alternative: Cloned reifications become useless for the client when they might be out-of-date with the object's model part [HLS00:ObInt], so that he will need to ask for a new clone. But cloning all the time, cloning objects with cycles in the representation structure, cloning distributed, multi-threaded objects, or cloning component objects "as large as whole applications" [Hop00:Com], whether for model parts or assistant candidates, can become very expensive both computationally and memory-wise.

Containment is a useful discipline for objects that are monomorphic and no collections. In open systems and systems with black-box reuse, the representative cannot trust externally created assistants more than they can trust external objects. What is then the sense of limiting the control of reference use to the assistants (P2)? A control of the exterior's use of references (P1) would associate usage restrictions only with references the exterior would not have obtained with containment in the first place. Once one gets into the business of controlling reference use, why should one stop half way? State control is the most flexible of the discussed disciplines, and still provides for object integrity. It allows most the uses of exposure. The only thing it still excludes is the mutation of returned assistants, which might be desirable in some cases where they reify model parts (public assistants), and the mutation of the representation through the iterator. Such mutations would not generally be safe w.r.t. modular representation reasoning. Read-only returned assistant are a compromise between flexibility and object integrity: It allows providing life-views of model parts, that are always up-to-date, but not model part manipulators.

A prerequisite for any automatic enforcement of object encapsulation by the compiler is to have a way to specify in the program where the abstraction boundaries are (P4). This specification alone is already useful: Knowing where the boundaries are allows to ensure that the object's model state and user-specified model

data concept (level)	abstract object (4)	implementation instance (3)	basic value variable (2)
representation	(nested) object composition	(nested) record of variables	bit-word
- size	dynamic	arbitrary, fixed	limited, fixed
- "glue"	references	contiguous memory	same memory cell
- defined	in design	through class module	by compiler
to prevent frontal breach	no cast to class type	modular name hiding	catch misinterpretation
to prevent lateral breach	... *alias control* ...	no pointers to variables	no up-calls from compiler

Figure 16: Data at different software levels

and representation invariants depend only on the representation and not on something else [MP01:Univ]. And only by knowing how much to abstract from, the actual semantics of an object, as defined by the program, can be inferred, and checked against the object's specification. (Checking makes no sense when the boundaries were reverse engineered with help of the specification, as in §5.2.)

6 Conclusions

We have seen how more abstract notions of data are reached by imposing an aggregation structure on data entities at a lower level and then abstracting from the aggregation's constituents. They become more and more powerful by the use of more and more flexible aggregation "glues" (see figures 1 and 16):

Aggregating the bits which are in the same memory cell (or register) yields bit-words which have a statically fixed size no larger than (double) processor words. Bit-words (in a memory cell) suffice to represent (variables of) basic concrete values like numbers, booleans, characters or references. Deciding the representation schemes of concrete values is usually not part of software development, but has been done much earlier by the implementors of the used high-level language.

Aggregating (variables of) basic values by special constructs like records, tuples, etc., yields compounds whose size is arbitrary, yet fixed during each compound's life-time.[23] Compounds suffice to represent 'simple" mutable abstract data entities like dates, complex numbers or lines as implementation instances. The decision on implementation instances' representation is encoded in one module as an implementation activity.

Aggregating implementation instances by references yields compositions which can grow and shrink dynamically. This provides the flexibility to represent arbitrary abstract data entities, or abstract objects, in an object-oriented context. Implementation instances "represent" abstract objects in two senses: groups of implementation instances are an abstract object's representation (reification) on a lower level, *and* one implementation instance is its representative (proxy). Finding the representation for an abstract object by decomposing it (top-down), or abstracting from a composition of implementation instances to an abstract object (bottom-up) are essential tasks in object-oriented design.

The second observation was that realizing the described advantages of abstract data stands and falls with the preservation of abstraction boundaries. A breach of abstraction can invalidate modular reasoning about (implementations and clients of) abstractions, destroy substitutivity and integrity of the abstraction's implementation, and thwarts independent reuse of implementations and clients of abstractions. It cannot be expected that all programmers working on large systems of programs from initial development to maintenance always have the right intuition about abstraction boundaries and respect them. Especially in system statically composed from commercial software components and in dynamically composed open component systems, one cannot rely on the other programmer to respect critical abstractions like security management.

Since Algol, type systems have been employed for automatically checking program code against declared intentions of abstraction to catch frontal abstraction breaches. First the interpretation of bit-words as concrete values was protected. "Progress in language design continued" [SG96:SAr] with modules which encapsulated the variables representing implementation instances. The latest step were abstract object types (Java interfaces), which are conceptually independent from implementation classes.

Compared to frontal breaches, abstraction breaches by type-correct(!) access to exposed internal parts of the representation (lateral breaches) have received only little attention. They are still manageable at the levels of values and of implementation instances. But for the object abstraction they are a serious threat with no simple solution. Several programming disciplines for object encapsulation have been proposed (containment, sandwiches, representation encapsulation, hiding and state control), but the current state of affairs is: Objects in general, unlike implementation instances, are *not* encapsulated, and thus using them is not guaranteed to always work as expected.

Typing object references by the type of the referred-

[23] Support for built-in dynamically resizing arrays seem to have become obsolete in, and consequentially cleaned off languages supporting user-defined data abstractions.

to object protected against frontal breaches. Protection against lateral breaches could be integrated into the framework of an object-oriented type system by additionally typing references by their function in the structure of object composition and in inter-object communication. However, current proposals of alias control techniques to this end are unsatisfactory, since they exclude some important practices of object-oriented software design (iterators, visitors, friendly functions, returning model part reifications) by too simple models of the structure of object representations, or by too conservative forms of encapsulation, or by both.

Unfortunately, object-oriented development methods do not spell out how to design a composition of objects so that the abstract object represented by it is effectively encapsulated. First, a deeper understanding will need to be gained of how object compositions translates to structures of references and what makes compositions of objects encapsulated. Only then can one hope to develop a useful system of reference types that enforces object encapsulation without excluding important object-oriented practices.

References

[ASS96:SICP] Harold Abelson, Gerald Jay Sussman, Julie Sussman: *Structure and Interpretation of Computer Programs* (2*nd.* ed.); MIT Press 1996.

[Alm97:Bal] Paulo Sérgio Almeida: *Balloon Types: Controlling Sharing of State in Data Types*; In *ECOOP'97*; LNCS 1241; Springer 1997.

[Ast96:ReoSpl] Hernán Astudillo R: *Reorganizing Split Objects*; 138-149 in *OOPSLA'96*; ACM 1996.

[Bac78:Fort] John Bachus: *The History of Fortran I, II, And III*; In Richard L Wexelblat (ed.): *History of Programming Languages*); Academic Press 1981.

[BLM97:HierEnc] J C Bicarregui, K C Lano, T S E Maibaum: *Objects, Associations and Subsystems: A Hierarchical Approach to Encapsulation*; 324-343 in *ECOOP'97*; LNCS 1241; Springer 1997.

[Bi+79:Simula] G M Birtwistle, O-J Dahl, B Myhrhaug, K Nygaard: *Simula begin* (2*nd.* ed.); Studentlitteratur, Bratt-Institut für Neues Lernen, and Cartwell-Brat 1980.

[Bis95:SuB] Norbert Bischof: *Struktur und Bedeutung: Eine Einführung in die Systemtheorie*; Verlag Hans Huber 1995.

[BC87:PartHier] Edwin Blake, Steve Cook: *On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk*; 41-50 in *ECOOP'87*; LNCS 276; Springer 1987.

[Boo94:OOAD] Grady Booch: *Object-Oriented Analysis and Design with Applications* (2*nd.* ed.); Addison-Wesley 1994. (first published 1991).

[Bro87:Sil] F P Brooks: *No Silver Bullet: Essence and Accidents of Software Engineering*; In *Computer* April 1987; IEEE 1987. First published in *Information Processing '86*; Elsevier 1986.

[Bru96:TyOOL] Kim B Bruce: *Typing in object-oriented languages: Achieving expressiveness and safety*; Technical Report; Williams College 1996.

[CLF93:OOSD] Dennis de Champeaux, Doug Lea, Penelope Faure: *Object-Oriented Software Development*; Addison-Wesley 1993.

[CLF92:POOD] Dennis de Champeaux, Doug Lea, Penelope Faure: *The Process of Object-Oriented Design*; 45-62 in *OOPSLA'92*; ACM 1992.

[Cha97:CSEM] David Chappell: *The Next Wave – Component Software Enters The Mainstream*; white-paper; Chappell & Associates 1997. http://www.rational.com/products/whitepapers/354.jsp

[Civ93:ComAD] Franco Civello: *Roles for composite objects in object-oriented analysis and design*; 376-393 in *OOPSLA'93*; ACM 1993.

[CPN98:OT] David G Clarke, John M Potter, James Nobel: *Ownership Types for Flexible Alias Protection*; 48-64 in *OOPSLA'98*; ACM 1998.

[DLN98:Wrest] David L Detlefs, K Rustan M Leino, Greg Nelson: *Wrestling with rep exposure*; SRC Research Report 156; DEC 1998.

[Eng01:Geh] Detlev Engler: *In geheimer Mission: Einhaltung des Geheimnisprinzips in Java*; 28-38 in *Java Spektrum* 1/2001; 101communications 2001.

[DL97:SpecInh] Krishna Kishore Dhara, Gary T Leavens: *Forcing behavioral subtyping through specification inheritance*; In *ICSE'96*; IEEE 1996. Also in a revised version as Technical Report TR#95-20c at Iowa State Univ., 1997.

[FM90:DSIO] J Fiadeiro, T Maibaum: *Describing, Structuring and Implementing Objects*; In *Foundations of Object-Oriented Languages*; LNCS 489; Springer 1991.

[Ga+95:DP] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns: Elements of Resuable Object-Oriented Software*; Addison-Wesley 1995.

[GR97:SMLBL] Emden R Gansner, John H Reppy (ed.): *Standard ML Basis Library proposal*; 1997. http://www.dina.kvl.dk/~sestoft/sml/

[GTZ98:Sand] Daniela Genius, Martin Trapp, Wolf Zimmermann: *An Approach to Improve Locality Using Sandwich Types*; In *Types In Compilation'98*; LNCS 1473; Springer 1998.

[GNU99:g++lib] GNU project: *The GNU Standard C++ Library*; header-files coming with gcc 2.95.2; FSF 1999.

[GR83:ST] Adele Goldberg, David Robson: *Smalltalk-80: The Language and its Implementation*; Addison-Wesley 1983.

[GJS00:Java] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: *The Java Language Specification* (2*nd.* ed.); Sun Microsystems 2000.

[GB99:OOFX] Aaron Greenhouse, John Boyland: *An Object-Oriented Effects System*; 205-229 in *ECOOP'99*; LNCS 1628; Springer 1999.

[HLS00:ObInt] Harri Hakonen, Ville Leppänen, Tapio Salakoski: *Object Integrity while Allowing Aliasing*; 91-96 in ICS'2000.

[Ham97:JB] Graham Hamilton (ed.): *JavaBeans* (1.01); Sun Microsystems 1997.

[Hop00:Com] Jon Hopkins: *Component Primer*; 27-30 in *Communications of the ACM* 43/10; ACM 2000.

[Hog91:Isl] John Hogg: *Islands: Aliasing Protection In Object-Oriented Languages*; In *OOPSLA'91*; ACM 1991.

[ISO98:C++] *Programming Languages – C++*; ISO/IEC 1998.

[ISO95:Ada] *Programming Languages – Ada*; ISO/IEC 1995.

[Ja+94:OOSE] Ivar Jacobson, Magnus Chrsterson, Patril Jonsson, Gunnar Övergaard: *Object-Oriented Software Engineering*; Addison-Wesley 1994. (first published 1992).

[Jen01:ASC] Tom Jennings: *ASCII: American Standard Code for Information Infiltration*; web-page;Word Power Systems 2001. http://www.wps.com/texts/codes/

[Ki+87:ComDB] Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F Garza, Darrell Woelk: *Composite Object Support in an Object-Oriented Database System*; In *OOPSLA'87*; ACM 1987.

[KT99:JAC] Günter Kniesel, Dirk Theisen: *JAC – Java with Transitive Readonly Access Control*; In should be in: Object-Oriented Technology: ECOOP'99 Workshop Reader, LNCS 1743, 1999.

[Lea99:LC++] Gary T Leavens: *Larch/C++ Reference Manual* (Revision 5.41); Iowa State Univ. 1999.

[Lei98:DG] K Rustan M Leino: *Data groups: Specifying the modification of extended state*; 144-153 in *OOPSLA'98*; ACM 1998.

[Lis92:CLU] Barbara H Liskov: *A History of CLU*; Technical Report; MIT 1992.

[LW94:BehNot] Barbara H Liskov, Jeanette M Wing: *A Behavioral Notion of Subtyping*; 1811-1841 in *TOPLAS* 16(1); ACM 1994.

[Lis88:DAH] Barbara H Liskov: *Data Abstractions and Hierarchy*; 17-34 in *SIGPLAN Notices* 25/5; ACM 1988.

[Lou94:PSp] Kenneth C Louden: *Programmiersprachen – Grundlagen, Konzepte, Entwurf*; Thomson 1994. Original title: *Programming Languages – Principles and Practice* 1993.

[Lyo89:EML] John Lyons: *Einführung in die moderne Linguistik* (7*th.* unchanged ed.); C H Beck 1989. – original: Introduction to Theoretial Linguistics, Cambridge Univ. Press 1968.

[Mar96:DIP] Robert C Martin: *The Dependency Inversion Principle*; In *C++ Report*; SIGS May 1996.

[Mee94:TaxDT] Brian Meek: *A taxonomy of datatypes*; 159-167 in *SIGPLAN Notices* 29/9; ACM 1994.

[Mey88:OOSC] Bertrand Meyer: *Object Oriented Software Construction*; Prentice Hall 1988.

[Min96:AlFr] Naftaly H Minsky: *Towards Alias-Free Pointers*; 189-209 in *ECOOP'96*; LNCS 1098; Springer 1996.

[MP01:Univ] Peter Müller, Arnd Poetzsch-Heffter: *Universes: A Type System for Alias and Dependency Control*; In *Informatik Berichte* 279; Fernuniversität Hagen 2001.

[MD95:COM] *The Component Object Model Specification* (0.9); Microsoft, DEC 1995.

[Mi+97:SML] Robin Milner, Mads Tofte, Robert Harper, David MacQueen: *The Definition of Standard ML (revised)*; MIT Press 1997.

[Nob99:OAl] James Noble: *The Objects of Aliasing*; presented at Intercontinental Workshop on Aliasing in Object Oriented Systems at ECOOP'99.

[NVP98:FAP] James Noble, Jan Vitek, John Potter: *Flexible Alias Protection*; In 158-185: *ECOOP'98*; LNCS 1445; Springer 1998.

[OR99:Aiken] John J O'Connor, Edmund F Robertson: *Howard Hathaway Aiken*; In *The MacTutor History of Mathematics archive*; web-page; Univ. of St. Andrews 1999. http://www-groups.dcs.st-and.ac.uk/~history/

[OMG00:CORBA] *The Common Object Request Broker: Architecture and Specification* (2.4); OMG 2000.

[OMG00:OCS] *Object Collection Specification* (1.0); OMG 2000. April

[OMG00:UML] *OMG Unified Modeling Language Specification* (1.3); OMG 2000.

[Pre97:KSF] Wolfgang Pree: *Komponentenbasierte Softwareentwicklung mit Frameworks*; dpunkt 1997.

[Res87:IlRe] Howard L Resnikoff: *The Illusion of Reality*; Spinger 1987.

[Rey94:DT/PD] John C Reynolds: *User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction*; In Carl A Gunter, John C Mitchell (ed.): *Theoretical Aspects of Object-oriented Programming: types, semantics, and language design*; MIT Press 1994. First published 1975.

[Rei92:Num] C Reinsch: *Numerische Mathematik I und II* (2. Auflage); Technische Universität München 1992.

[Rum97:Myths] James Rumbaugh: *OO Myths: Assumptions from a language view*; In *JOOP* 9(9); SIGS 1997.

[Rum94:ComBB] James Rumbaugh: *Building boxes: Composite objects*; In *JOOP* 7(7); SIGS 1994.

[SIP97:SigBug] Secure Internet Programming Group: *HotJava 1.0 Signature Bug*; web-page; Princeton Univ. 1997. http://www.cs.princeton.edu/sip/news/april29.html

[Set97:PLCC] Ravi Sethi: *Programming Languages: Concepts & Constructs* (2nd. ed.); Addison-Wesley 1997.

[SG96:SAr] Mary Shaw, David Garlan: *Software Architecture: perspectives on an emerging discipline*; Prentice-Hall 1996.

[Sny86:EncInh] Alan Snyder: *Encapsulation and Inheritance in Object-Oriented Programming Languages*; 38-45 in *OOPSLA'86*; ACM 1986.

[Sou97:TaC] Desmon d'Souza: *Types and classes: A language-independent view*; In *JOOP* 10(1); SIGS 1997.

[Sta97:MSC] Raymie Stata: *Modularity in the Presence of Subclassing*; Research Report 145; DEC/SRC 1997.

[Str87:WIOOP] Bjarne Stroustrup: *What is "Object-Oriented Programming"?*; 51-70 in *ECOOP'87*; LNCS 276; Springer 1987.

[SM97:ThCOM] Kevin J Sullivan, Mark Marchukov: *Interface Negotiation and Efficient Reuse: A Relaxed Theory of the Component Object Model*; CS-97-11; Univ. of Virginia 1997.

[Sun00:JavaAPI] *Java 2 Platform API Specification* (Standard Edition, v 1.3); Sun Microsystems 2000.

[Thu97:Z3] A Thurm: *Die Rechenmaschine Z3 von Konrad Zuse (1910-1995)*; web page; 1997. http://www.histech.rwth-aachen.de/www/quellen/Z3Emulator/

[Tur51:Man] A M Turing: *A Programmers' Handbook for the Manchester Electronic Computer Mark II*; 1951. Available at http://www.computer50.org/kgill/mark1/mark1book.html

[Un+91:PWC] David Ungar, Craig Chambers, Bay-Wei Chang, Urs Hölzle: *Organizing Programs Without Classes*; In *Lisp and Symbolic Computation* 4(3); Kluwer 1991.

[US87:Self] David Ungar, Randall B Smith: *Self: The Power of Simplicity*; 227-241 in *OOPSLA'87*; ACM 1987.

[Uni00:Unicode] The Unicode Consortium et.al.: *The Unicode Standard, Version 3.0*; Addison-Wesley 2000.

[VMO99:ESCO] S Vauttier, M Magan, C Oussalah: *Extended Specification of Composite Objects in UML*; In *JOOP* 12(2); SIGS 1999.

[Wil92:FMOOP] Alan Cameron Wills: *Formal Methods applied to Object-Oriented Programming*; PhD thesis; Univ. Manchester 1992.

[Win83:Char] Johan W van Wingen: *Character sets. LETTERS, TOKENS and CODES*; web page; 1999. http://www.terena.nl/multiling/euroml/mlcs5.html

[Wir83:Mod2] Niklaus Wirth: *Programming in Modula-2* (2nd. ed.); Springer 1983.

A Specifications and Programs

Specification of Date

Below is the full specification of data abstraction `Date` (c.f. §4.1), followed by auxiliary specification functions. $\alpha ?\ \beta : \gamma$ is the conditional expression, and $n\%m$ the modulo expression of Java.

```
data abstraction class Date {
    spec d  = public final int dayOfMonth();
    spec m  = public final int month();
    spec y  = public final int year();
    public Date(int dd, int mm, int yy)
            requires isdate(dd,mm,yy);
            ensures  d=dd ∧ m=mm ∧ y=yy;
    public void Advance()
            ensures d  = old dayOfNext(d,m,y)
                  ∧ m  = old monthOfNext(d,m,y)
                  ∧ y  = old yearOfNext(d,m,y);
    public int dayOfYear()
            ensures result = d + daysInPrevMonths(y,m);
    public boolean sameday(Date d2)
            ensures result = (d = d2.d ∧ m = d2.m ∧ y = d2.y);
    public Date next()
            ensures result.d  = dayOfNext(d,m,y)
                  ∧ result.m  = monthOfNext(d,m,y)
                  ∧ result.y  = yearOfNext(d,m,y);
    public Date prev()
           ensures d  = dayOfNext(result.d,result.m,result.y)
                 ∧ m  = monthOfNext(result.d,result.m,result.y)
                 ∧ y  = yearOfNext(result.d,result.m,result.y);
}
boolean isleap(int y) =
        y%4=0 ∧ (y%200≠0 ∨ y%1000=0);
int daysInMonth(int y, int m) =
        m=1? 0 : m ∈ {4,6,9,11}? 30 : 31;
boolean isdate(int d, int m, int y) =
        0 < m≤12 ∧ 0 < d≤daysInMonth(y,m);
int daysInPrevMonths(int y, int m) =
          m=1?    0
        : m=2?    31
        : m=3?    (isleap(y)?  59:58)     — + 28/27
        : m=4?    (isleap(y)?  90:89)     — + 31
        : m=5?    (isleap(y)? 120:119)    — + 30
        : m=6?    (isleap(y)? 151:150)    — + 31
        : m=7?    (isleap(y)? 181:180)    — + 30
        : m=8?    (isleap(y)? 212:211)    — + 31
        : m=9?    (isleap(y)? 243:242)    — + 31
        : m=10?   (isleap(y)? 273:272)    — + 30
        : m=11?   (isleap(y)? 304:303)    — + 31
        :         (isleap(y)? 334:333);   — + 30
int dayOfNext(int d, int m, int y) =
        isdate(d+1,m,y)?  d+1 : 1;
int monthOfNext(int d, int m, int y) =
        isdate(d+1,m,y)?  m : m<12? m+1 : 1;
int yearOfNext(int d, int m, int y) =
        isdate(d+1,m,y)?  y : m<12? y : y+1;
```

Three Integer Variant of Class Date

This version of class `Date` uses private instance variables `d`, `m` and `y` to represent the corresponding model variables. Hence model invariant isdate(d,m,y) translates one-to-one to representation invariant isdate(`d`,`m`,`y`).

```
class Date {
  private int y, m, d;
  public Date(int dd, int mm, int yy)
                             { d=dd; m=mm; y=yy; }
  public void Advance()
                        { d = dayOfNext(d,m,y);
                          m = monthOfNext(d,m,y);
                          y = yearOfNext(d,m,y);  }
  public int year()                    { return y; }
  public int month()                   { return m; }
  public int dayOfMonth()              { return d; }
  public int dayOfYear()
                { return d+daysInPrevMonths(y,m); }
  public boolean sameday(Date d2)
          { return d==d2.d && m==d2.m && y==d2.y; }
  public Date next()
            { return new Date(dayOfNext(d,m,y),
                              monthOfNext(d,m,y),
                              yearOfNext(d,m,y)); }
  public Date prev() ...
  // auxiliary functions
  private static boolean isleap(int y) ...
  private static int daysInMonth(int y, int m) ...
  ...
}
```

The operations are implemented with the help of private static functions. Their implementation can be derived trivially from the definition of the specification functions with the respective name. Implementing `prev` is left as an exercise to the reader.

Two Integer Variant of Class Date

The auxiliary private static functions of class `Date` version in §4.2 are given below. Functions corresponding to specification functions (see above) are not shown.

```
private static int daysInYear(int y)
                { return isleap(y)? 355:354; }
private static int monthOf(int y, int d)
         { return d <= 31?                   1
                : d <= (isleap(y)?  59:58)?   2
                : d <= (isleap(y)?  90:89)?   3
                : d <= (isleap(y)? 120:119)?  4
                : d <= (isleap(y)? 151:150)?  5
                : d <= (isleap(y)? 181:180)?  6
                : d <= (isleap(y)? 212:211)?  7
                : d <= (isleap(y)? 243:242)?  8
                : d <= (isleap(y)? 273:272)?  9
                : d <= (isleap(y)? 304:303)? 10
                : d <= (isleap(y)? 334:333)? 11
                :                            12;
         }
private static int dayOf(int y, int d)
  { return d - daysInPrevMonths(y, monthOf(y,d)); }
```

Class Node

Each instance of class `Node` has a pointer *el* to an element object, and a pointer *nx* to the next node. Constructor `Node` creates nodes with *el* and *nx* initialized to `null`. There are observers to get, and mutators to set *el* and *nx*. The implementation in class module `Node` is straightfoward.

```
data abstraction class Node {
   spec el  = public final Object elem();
   spec nx  = public final Node   next();
   public Node() ensures el=null ∧ nx=null;
   public void SetElem(Object x)
               ensures  el=x ∧ nx=old nx;
   public void SetNext(Node n)
               ensures  el=old el ∧ nx=n;
 }
```

```
class Node {
  private Object  el;
  private Node    nx;
  public        Node()     { el=null; nx=null; }
  public Object elem()              { return el; }
  public Node   next()              { return nx; }
  public void   SetElem(Object x)       { el=x; }
  public void   SetNext(Node n)         { nx=n; }
}
```

Class SetImpl

This is class `SetImpl` as specified in §5.2:

```
class SetImpl[Ob] implements Set[Ob] {
  private Node[Ob] anchor;

  public  SetImpl() { anchor=null; }

  public  boolean has(Ob x)
  {  if(anchor==null) return false;
     Node[Ob] n = anchor;
     do {  if(n.elem()==x) return true;
           n = n.next();
     } while(n!=anchor)
     return false;
  }
  public  void Insert(Ob x)
  {  Node[Ob] n = new Node[Ob];
     n.SetElem(x);
     if(anchor==null) anchor = n;
     else             n.SetNext(anchor.next());
     anchor.SetNext(n);
  }
  public  void Remove(Ob x)
  {  if(anchor==null) return;
     anchor = first_without(anchor,anchor,x);
     if(anchor==null) return;
     Node[Ob] n = anchor;
     do {  // note: at least anchor is a node without x
           while(n.next()!=null && n.next().elem()==x)
              n.SetNext(n.next().next());
           n = n.next(); // next does not have x!
     } while(n!=anchor);
  }
  private static  Node[Ob]
  first_without(Node[Ob] n, Node[Ob] stop, Ob x)
  {  return n==stop?      null
          : n.elem()!=x?  n
          : first_without(n.next(),stop,x);
  }
} // end class SetImpl
```