The C Object System *

Using C as a High-Level Object-Oriented Language

Laurent Deniau
CERN – European Organization for Nuclear Research
laurent.deniau@cern.ch

Abstract
The C Object System (COS) is a small C library which implements high-level concepts available in CLOS, OBJECTIVE-C and other object-oriented programming languages: uniform object model (class, metaclass and property-metaclass), generic functions, multi-methods, delegation, properties, exceptions, contracts and closures. COS relies on the programmable capabilities of the C programming language to extend its syntax and to implement the aforementioned concepts as first-class objects. COS aims at satisfying several general principles like simplicity, extensibility, reusability, efficiency and portability which are rarely met in a single programming language. Its design is tuned to provide efficient and portable implementation of message multi-dispatch and message multi-forwarding which are the heart of code extensibility and reusability. With COS features in hand, software should become as flexible and extensible as with scripting languages and as efficient and portable as expected with C programming. Likewise, COS concepts should significantly simplify adaptive and aspect-oriented programming as well as distributed and service-oriented computing.

Categories and Subject Descriptors D.3.3 [C Programming Language]: Language Constructs and Features; D.1.5 [Programming Techniques]: Object-oriented Programming.

General Terms Object-oriented programming.

Keywords Adaptive object model, Aspects, Class cluster, Closure, Contract, Delegation, Design pattern, Exception, Generic function, Introspection, High-order message, Message forwarding, Meta class, Meta-object protocol, Multi-method, Open class model, Predicate dispatch, Programming language design, Properties, Uniform object model.

1. Motivation
The C Object System (COS) is a small framework which adds an object-oriented layer to the C programming language [1, 2, 3] using its programmable capabilities while following the simplicity of OBJECTIVE-C [5, 6] and the extensibility of CLOS [8, 9, 10]. COS aims to fulfill several general principles rarely met in a single programming language: simplicity, extensibility, reusability, efficiency and portability.

1.1 Context
COS has been developed in the hope to solve fundamental programming problems encountered in scientific computing and more specifically in applied metrology [11, 12]. Although this domain looks simple at first glance, it involves nonetheless numerous fields of computer science; from low-level tasks like the development of drivers, protocols or state machines, the control of hardware, the acquisition of data, the synchronization of concurrent processes, or the numerical analysis and modeling of huge data sets; to high-level tasks like the interaction with databases or web servers, the management of remote or distributed resources, the visualization of complex data sets or the interpretation of scripts to make the system configurable and controllable by non-programmers [13, 14, 15]. Not to mention that scientific projects commonly have to rely on sparse human resources to develop and maintain for the long term such continually-evolving-systems (i.e. R&D). Therefore the challenge is ambitious but I firmly believe that COS provides the required features to simplify the development and the support of such systems as well as a wide variety of software projects.

1.2 Principles
Given the context, it is essential to reduce the multiplicity of the technologies involved, to simplify the development process, to enhance the productivity, to guarantee the extensibility and the portability of the code and to adapt the required skills to the available resources. Hence, the qualities of the programming language are essential for the success of such projects and should focus on the following principles:

* COS project: http://sourceforge.net/projects/cos

Copyright notice will appear here once 'preprint' option is removed.

Draft research paper for OOPSLA’09

1 In the sense of “Lisp is a programmable programming language”, [4].
Simplicity  The language should be easy to learn and use. The training curve for an average programmer should be as short as possible what implies in particular a clear and concise syntax. Simplicity should become an asset which guarantees the quality of the code and allows to write complex constructions without being penalized by a complex formalism or by the multiplicity of the paradigms. COS can be learned within a few days by C programmers with some knowledge of object-oriented concepts, although exploiting the full power of COS requires some experience.

Extensibility  The language should support the addition of new features or the improvement of existing features without changing significantly the code or the software architecture. Concepts like polymorphism, message dispatch and open class model help to achieve good flexibility and extensibility by reducing coupling. But they usually have a strong impact on the efficiency. COS dispatches messages with an efficiency in the range of the C++ virtual member functions.

Reusability  The language should support code reusability, namely the ability to reuse or quickly adapt existing components to unforeseen tasks. It is easier to achieve this goal if the language allows to write generic code, either by parameterization, either by abstraction, to ease the componentization of design patterns [16, 17, 18]. To support the development of generic components, COS provides multi-methods to handle dynamic and polymorphic collaboration and delegation to handle dynamic and polymorphic composition.

Efficiency  A general purpose programming language must be efficient, that is it must be able to translate all kinds of algorithms into programs running with predictable resource usage (mainly CPU and memory) consistent with the processes carried out. In this respect, programming languages with an abstract machine close to the physical machine — a low-level language — offer generally better results. C is admittedly known to achieve good efficiency.

Portability  A general purpose programming language must be portable, that is it must be widely available on many architectures and it must be accessible from almost any other languages (F#). This point often neglected brings many advantages: it improves the software reliability, it reduces the deployment cost, it enlarges the field of potential users and it helps to find trained programmers. Regarding this point, normalized programming languages (ISO) get the advantage. ISO C99 is normalized and well known for its availability and portability.

1.3  Proposition

COS extends the C programming language with concepts [19] mostly borrowed from OBJECTIVE-C and CLOS. The choice of designing the language as a C library instead of a compiler allowed to quickly explore various object models, but R.E. Johnson’s paper on the dynamic object model [20] definitely focused my research towards the final design:

“If a system is continually changing, or if you want users to be able to extend it, then the Dynamic Object Model architecture is often useful. [...] Systems based on Dynamic Object Models can be much smaller than alternatives. [...] I am working on replacing a system with several millions lines of code with a system based on a dynamic object model that I predict will require about 20,000 lines of code. [...] This makes these systems easier to change by experts, and (in theory) should make them easier to understand and maintain. But a Dynamic Object Model is hard to build. [...] A system based on a Dynamic Object Model is an interpreter, and can be slow.”

This adaptive object model [21, 22] is actually what COS provides, but at the level of the C programming languages without significant efficiency loss. In particular, COS has been designed to support efficiently two key concepts — multi-methods and fast generic delegation — and provides a uniform object model where classes, generics and methods are first-class objects. Incidentally, COS strengthens inherently all the guidelines stated in [23] to build “flexible, usable and reusable object-oriented frameworks” as well as architectural pattern proposed in [24] to design flexible component-based frameworks.

2. Overview

COS is a small framework entirely written in portable C99 which provides programming paradigms like objects, classes, metaclasses, generic functions, multi-methods, delegation, properties, exceptions, contracts and closures. COS syntax and features are directly available at the C source code level through the use of the language keywords defined in the header file <cos/Object.h>.

2.1  Concepts

Polymorphism  This concept available in object-oriented programming languages is the heart of software extensibility because it postpones to runtime the resolution of methods invocation and reduces coupling between callers and callees. Besides, if the polymorphic types are dynamic, the coupling becomes almost inexistent and code size and complexity are significantly reduced. On one hand, these simplifications usually improve the programmer understanding who makes less conceptual errors, draws simpler designs and increases its productivity. On the other hand, dynamic typing postpones the detection of unknown messages at runtime, with the risk to see programs ending prematurely. But well tested software reduce this risk to exceptional situations.

Collaboration  Software development is mainly about building collaborations between entities, namely objects. As soon as polymorphic objects are involved everywhere to ensure good software extensibility and reusability, one needs polymorphic collaboration implemented by multi-methods. They

---

2  Namely C89 and C99 variadic macros.
reduce strong coupling that exist in the Visitor pattern (or equivalent) as well as the amount of code needed to achieve the task. COS provides message multi-dispatch with an efficiency in the range of the C++ virtual member function.

**Composition** The composition of objects and behaviors is a well known key-concept in software design. It enhances software flexibility by introducing levels of indirection in objects and behaviors. Most structural and behavioral design patterns described in [28] introduce such indirections, but at the price of an increased code complexity and coupling and hence a decreased reusability of the components built. The delegation is an effective mechanism which allows to manage the composition of both, objects and behaviors, without introducing coupling. COS provides delegation with the efficiency of message dispatch, seemingly a unique feature.

**Reflection** Reflection is a powerful aspect of adaptive object models which, amongst others, allows to mimic the behavior of interpreters. COS provides full introspection and limited intercession on polymorphic types and behaviors, that is classes, generics and methods, as well as object attributes through the definition of properties. Since all COS components are *first-class objects*, it is trivial to replace creation-patterns [28] by generic functions (section 8.1).

**Encapsulation** Encapsulation is a major concern when developing libraries and large-scale software. COS enforces encapsulation of class implementation because encapsulation is not only a matter of managing coupling but also a design issue. Besides, the object behaviors are represented by generics which favors the separation of concerns of interfaces and reduces cross-interfaces dependencies [23]. Moreover, the open class model of COS allows to extend classes on need without breaking the encapsulation (i.e. without “re-opening the box”) and reduces the risk of premature design.

**Ownership** The management of object life cycles requires a clear policy of ownership and scope rules. In languages like C and C++ where semantic by value prevails, the burden is put on the programmer’s shoulders. In languages like JAVA, C# and D where semantic by reference prevails, the burden is put on the garbage collector. In this domain, COS lets the developer choose between garbage collection (*e.g.* Boehm GC [25]) and manual reference counting with rich semantic (section 3.5).  

**Concurrency** COS has been designed from the beginning with concurrency in mind and shares only its dictionary of static components. Per thread resources like message caches and autorelease pools rely on either thread-local-storage or thread-specific-key according to the availability.

### 2.2 Components

The object-oriented layer of COS is based on three components (figure 1) borrowed from CLOS which characterize the *open object model* described in depth in [8] and [9].

<table>
<thead>
<tr>
<th><strong>Function/Behavior/Verb</strong></th>
<th><strong>Type/State/Noun</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>defgeneric</td>
<td>structure definition</td>
</tr>
<tr>
<td>function declaration</td>
<td>(interface)</td>
</tr>
<tr>
<td>defclass</td>
<td>function definition</td>
</tr>
<tr>
<td>function declaration</td>
<td>(implementation)</td>
</tr>
</tbody>
</table>

**Figure 1.** Roles of COS components and their equivalent C-forms. Multi-methods are classes specialization of generics.

#### Classes

Classes play the same role as *structures* in C and define object *attributes*. They are bound to their *superclass* and *metaclasses* and define supertypes-subtypes hierarchies.

#### Generics

Generics play the same role as *function declarations* in C and define *messages*. They are essential actors of code extensibility and ensure correctness of formal parameters of messages between caller and callee.

#### Methods

Methods play the same role as *function definitions* in C and define *specializations* of generics. A method is invoked if the message belongs to its generic and the receivers match its classes (multi-methods).

The similarities between COS components and their equivalent C-forms let C programmers with some notions of object-oriented design be productive rapidly. The open object model allows to define components in different places and therefore requires an extra linking iteration to collect their *external symbols*: link → collect⁴ → re-link. This fast iteration is automatically performed by the makefiles coming with COS before the final compilation stage that builds the executable or the dynamic library.

### 2.3 Syntax

COS introduces new keywords to extend the C language with a user-friendly syntax half-way between OBJECTIVE-C and CLOS. COS parses its syntax and generates code with the help of its functional C preprocessing library³; a module of a few hundred C macros which was developed for this purpose. It offers limited parsing capabilities, token recognition, token manipulation and algorithms like eval, map, filter, fold, scan, split borrowed from functional languages and working on tuples of tokens. As a rule of thumb, all COS symbols and macros are mangled to avoid unexpected collisions with other libraries, including keywords which can be disabled.

Despite of its dynamic nature, COS tries hard to detect all syntax errors, type errors and other mistakes at compile-

³ COS mangled symbols are collected with the nm command or equivalent.

⁴ The description of this *cpp* library is beyond the scope of this paper.
time by using static asserts or similar tricks and to emit meaningful diagnostics. The only point that C\textsc{os} cannot check at compile time is the understanding of a message by the receivers; an important “feature” to reduce coupling.

The syntax and grammar of C\textsc{os} are summarized in the figures 2, 7, 8, 9, 10 and 11, following the notation of the C99 standard \[1\].

3. Classes (nouns)

C\textsc{os} allows to define and use classes as easily as in other object-oriented programming languages.

3.1 Using classes

The \texttt{useclass()} declaration allows to access to classes as first-class objects. The following simple program highlights the similarities between C\textsc{os} and OBJECTIVE-C:

```c
#include <cos/Object.h>
#include <cos/generics.h>

useclass(Counter, (Stdout)out);

int main(void) {
    OBJ cnt = gnew(Counter);
    gput(out,cnt);
    gdelete(cnt);
}
```

which can be translated line-by-line into OBJECTIVE-C by:

```objc
#include <objc/Object.h>

@class Counter, Stdout;

int main(void) {
    id cnt = [Counter new];
    [Stdout put: cnt];
    [cnt release];
}
```

Lines 7 – 9 show the life cycle of objects, starting with \texttt{gnew} (resp. \texttt{new}) and ending with \texttt{gdelete} (resp. \texttt{release}). They also show that generics are functions (e.g. one can take their address). Finally, the line 8 shows an example of multi-method where the message \texttt{gput(\_\_\_)} will look for the specialization \texttt{gput(mStdout,Counter)} whose meaning is discussed in section 5. In order to achieve the same task, OBJECTIVE-C code has to rely on the Visitor pattern, a burden that requires more coding, creates static dependencies (strong coupling) and is difficult to extend.

3.2 Defining classes

The definition of a class is very similar to a C structure:

```c
defclass(Counter)
    int cnt;
endclass
```

which is translated in OBJECTIVE-C as:

```objc
@interface Counter : Object {
    int cnt;
}
@end
```

or equivalently in C\textsc{os} as:

```
useclass( Counter );
class-declaration:
    useclass( class-decl-list );
class-decl-list:
    class-decl , class-decl-list

class-decl:
    class-name
    ( class-name ) local-name

class-definition:
    defclass ( class-specifier )
    struct-declaration-list (c99)
    endclass

class-instantiation:
    makclass ( class-specifier );

class-specifier:
    class-name
    ( class-name ) super-class-name

{ class, super-class, local }-name:
    identifier (c99)
```

Figure 2. Syntax summary of classes.

---

\[1\] By convention, the name of generics always starts by a ‘g’.

\[2\] By convention, the name of classes always starts by an uppercase letter.
Counter

What must be visible and when? In order to manage coupling, COS provides three levels of visibility: none, declaration and definition. If you only use the generic type OBJ, nothing is required (no coupling):

```c
OBJ gnew(OBJ cls) {
    return ginit(galloc(cls));
}
```

If you want to create instances of a class, only the declaration is required (weak coupling):

```c
OBJ gnewBook(void) {
    useclass(Book); // local declaration
    return gnew(Book);
}
```

If you want to define subclasses, methods or instances with automatic storage duration, the class definition must be visible (strong coupling).

### 3.3 Class inheritance

Class inheritance is as easy in COS as in other object-oriented programming languages. Figure 3 shows the hierarchies of the core classes of COS deriving from the root classes Object and Nil. As an example, the MilliCounter class defined hereafter derives from the class Counter to extend its resolution to thousandths of count:

```c
(defclass MilliCounter (Counter) ((mcnt)) )
```

This definition of the attribute default behavior when the superclass isn’t specified — and the automatic storage duration, the class definition must be visible.

### 3.4 Meta classes

Like in OBJECTIVE-C, a COS class definition creates a parallel hierarchy of metaclass which facilitates the use of classes as first-class objects. Figure 4 shows the complete hierarchy of the PropMetaClass class, including its metaclasses.

#### Class metaclass

The metaclasses are classes of classes implicitly defined in COS to ensure the coherency of the type system: to each class must correspond a metaclass [26]. Both inheritance trees are built in parallel: if a class A derives from a class B, then its metaclass mA derives from the metaclass mB — except the root classes which derive from Nil and have their metaclasses deriving from Class to close the inheritance path. Metaclasses are instances of the class MetaClass.

In the three cases, the derived class inherits the attributes and the methods of its superclass. Since COS aims at insulating classes as much as possible, it discourages direct access to superclass attributes by introducing a syntactic indirection which forces the user to write `obj->Super.attribute` instead of `obj->attribute`. The inheritance of multi-methods has a different meaning and will be discussed in section 5.

#### Root class

Defining a root class is an exceptional task but it may be a necessity in some rare cases. COS uses the terminal symbol \( \perp \) (represented by ‘\(_\)’) to declare a class as a root class. For example, Object is an important root class with the following simple definition:

```c
(defclass (Object,) )
```

But its methods must be defined with care since they provide all the essential functionalities inherited by other classes.

#### Class rank

COS computes at compile-time the inheritance depth of each class. The rank of a root class is zero (by definition) and each successive subclass increases the rank.

#### Dynamic inheritance

COS provides the message `gchange-class(obj.cls)` to change the class of `obj` to `cls` iff it is a superclass of `obj`’s class; and the message `gunsafeChange-class(obj.cls, spr)` to change the class of `obj` to `cls` iff both classes share a common superclass `spr` and the instance size of `cls` is lesser or equal to the size of `obj`. These messages are useful for implementing class clusters, state machines and adaptive behaviors.

---

7 \( \perp \) means “end of hierarchy” or \texttt{Nil}, but not the class \texttt{Nil}.

8 The metaclass name is always the class name prefixed by a ‘m’.
Property metaclass  In some design patterns like Singleton or Class Cluster, or during class initialization (section 3.6), the automatic derivation of the class metaclass from its superclass metaclass can be problematic as detailed in [27]. To solve the problem, COS associates each class a property metaclass which cannot be derived; that is all methods specialized on the property metaclass can only be reached by the class itself. In order to preserve the consistency of the hierarchy, a property metaclass must always derive from its class metaclass, namely pmA (resp. pmB) derives from mA (resp. mB) as shown in the figure 4. Property metaclasses are instances of the class PropMetaClass.

Class objects  With multi-methods and metaclasses in hands, it is possible to use classes as common objects. Figure 5 shows the hierarchy of the core class-objects used in COS to specialized multi-methods with specific states. For instance messages like gand, gor and gnot are able to respond to messages containing the class-predicates True, False and TrueFalse. The root class Nil is a special class-object which means no-object but still safe for message dispatch: sending a message to Nil is safe, but not to NIL.

Type system  The COS type system follows the rules of Objective-C, that is polymorphic objects have opaque types (ADT) outside their methods and are statically and strongly typed inside; not to mention that multi-methods reduce significantly the need for runtime identification of polymorphic parameters. Furthermore, the set of class – metaclass – property-metaclass forms a coherent hierarchy of classes and types which offers better consistency and more flexibility than in Objective-C and Smalltalk where metaclasses are not explicit and derive directly from Object.

Figure 4. COS core classes hierarchy with metaclasses.

Figure 5. Subset of COS core class-predicates hierarchy.

3.5 Class instances

Object life cycle  The life cycle of objects in COS is very similar to other object-oriented programming languages, namely it starts by creation (galloc) followed by initialization (ginit and variants) and ends with deinitialization (gdeinit) followed by destruction (gdealloc). In between, the user manages the ownership of objects (i.e. dynamic scope) with gretain, gorelease and gautoRelease like in Objective-C. The copy initializer is the specialization of the generic ginitWith(_,-) for the same class twice. The designated initializer is the initializer with the most coverage which invokes the designated initializer of the superclass using next_method. Other initializers are secondary initializers which must invoke the designated initializer [7].

Object type  In COS (resp. Objective-C), objects are always of dynamic type because the type of galloc (resp. alloc) is OBJ (resp. id). Since it is the first step of the life cycle of objects in both languages, the type of objects can never be known statically, except inside their own multi-methods. That is why COS (resp. Objective-C) provides the message gisKindOf(obj,cls) (resp. [obj isKindOfClass: cls]) to inspect the type of objects. But even so, it would be dangerous to use a static cast to convert an object into its expected type because dynamic design patterns like Class Cluster and Proxy might override gisKindOf for their use. COS also provides the message gclass(obj) which returns obj’s class.

Object identity  In COS, an object is bounded to its class through a unique 32-bit identifier produced by a linear congruential generator which is also a generator of the cyclic groups \( \mathbb{Z}/2^k\mathbb{Z} \) for \( k = 2..32 \). This powerful algebraic property allows to retrieve efficiently the class of an object from the components table using its identifier as an index (Figure 6). Comparing to pointer-based implementations, the unique identifier has four advantages:

It ensures better behavior of cache lookups under heavy load (uniform hash), it makes the hash functions very fast (sum of shifted ids), it is smaller than pointers on 64-bit
Dictionary of behavioral components

\[
\text{obj->id} \rightarrow \text{obj's class}
\]

\[
\text{table[obj->id & (table_size-1)]}
\]

**Figure 6.** Lookup to retrieve object’s class from object’s id.

**Automatic objects** Since COS adds an object-oriented layer on top of the C programming language, it is possible to create objects with automatic storage duration (e.g. on the stack) using compound literals (C99). In order to achieve this, the class definition must be visible and the developer of the class must provide a special constructor. For example the constructor `aStr("a string")` is equivalent to the `OBJECTIVE-C` directive `@a "a string"`. COS already provides automatic constructors for many common objects like Char, Short, Int, Long, Float, Complex, Range, Functor and Array. Automatic constructors allow to create efficiently temporary objects with local scope and enhance the flexibility of multi-methods. For example, the initializer `ginitWith(,,,)` and its variants can be used in conjunction with almost all the automatic constructors aforementioned. Thanks to the rich semantic of COS reference counting, if an automatic object receives the message `gretain` or `gautoDelete`, it is automatically cloned using the message `gclone` and the new copy with dynamic scope is returned.

**Static objects** Static objects can be built in the same way as automatic objects except that they require some care in multi-threaded environments. It is worth to note that all COS components have static storage duration and consequently are insensitive to ownership and cannot be destroyed.

### 3.6 Implementing classes

Class instantiations create the `class` objects using the keyword `makclass` and the same `class-specifier` as the corresponding `defclass`. COS checks at compile-time if both definitions match. The counters implementation follows:

```c
makclass (Counter);
makclass (MilliCounter,Counter);
```

which is equivalent in `OBJECTIVE-C` to:

```c
@implementation Counter
// definition of Counter methods not shown
@end
@implementation MilliCounter: Counter
// definition of MilliCounter methods not shown
@end
```

By convention, automatic constructors always start by an ’a’.

**Class initialization** For the purpose of pre-initialization, COS ensures to invoke **once by ascending class rank** (superclass first) all specializations of the message `ginitialize` on `property` metaclass before the first message is sent. Likewise, COS ensures to invoke **once by descending class rank** (subclasses first) all specializations of the message `gdeinitialize` on `property` metaclass after exiting `main`.

### 4. Generics (verbs)

We have already seen in previous code samples that generics can be used as functions. But generics take in fact multiple forms and define each:

- a function declaration (`defgeneric`) which ensures the correctness of the signature of its methods (`deffirstmethod`), aliases (`defalias`) and next-methods (`deffirstmethod`).
- a function definition used to dispatch the message and to find the most specialized method belonging to the generic and matching the classes of the receivers.
- an object holding the generic’s metadata: the selector.

A generic function has one definition of its semantics and is, in effect, a verb raised at the same level of abstraction as a noun [4]. Figure 7 summarizes the syntax of generics, half way between the syntax of generic’s definition in CLOS and the syntax of method’s declaration in `OBJECTIVE-C`.

**Generic rank** The rank of a generic is the number of receivers in its `param-list`. COS supports generics from rank 1 to 5 what should be enough in practice since rank 1 to 4 already cover all the multi-methods defined in the libraries of Cecil and Dylan [29, 30, 36].

#### 4.1 Message dispatch

COS dispatch uses global caches (one per generics rank) implemented with hash tables to speedup method lookups. The caches solve slot collisions by growing until they reach a configurable upper bound of slots. After that, they use packed linked list incrementally built to hold a maximum of 3 cells. Above this length, the caches start to forget cached methods — a required behavior when dynamic class creation is supported. The lookup uses fast asymmetric hash functions (sum of shifted ids) to compute the cache slots and ensures uniform distribution even when all selectors have the same type or specializations on permutations exist.

**Fast messages** COS lookup is simple enough to allow some code inlining on the caller side to speedup message dispatch. Fast lookup is enabled **up to the generic rank** specified by `COS_FAST_MESSAGE` — from disabled (0) to all (5, default) — before the generic definitions (defgeneric).

#### 4.2 Declaring generics

Generic declarations are less common than class declarations but they can be useful when one wants to use generics as first-class objects. Since generic definitions are more
4.3 Defining generics

Definitions of generics correspond to function declarations in C and differ from OBJECTIVE-C method declarations by the fact that they are neither bound to classes (prefix ‘-‘) nor to metaclasses (prefix ‘+‘). The following definitions:

```c
defgeneric(void, gincr, _1); // rank 1
defgeneric(void, gincrBy, _1, (int)by); // rank 1
defgeneric(OBJ, ginitWith, _1, _2); // rank 2
defgeneric(OBJ, ggetAt, _1, at); // rank 2
defgeneric(void, gputAt, _1, at, what); // rank 3
```

can be translated into CLOS as:

```lisp
(defgeneric incr (obj))
(defgeneric incr-by (obj by))
(defgeneric initWith (obj with))
(defgeneric get-at (obj at))
(defgeneric put-at (obj at what))
```

Selector parameters like `at` are called *open types* (no parenthesis) since their type can vary for each specialization. Other parameters like `by` are called *closed types* (with parenthesis) and have fixed types and names: specializations must use the same types and names as defined by the generic. This enforces the semantic of *monomorphic* parameters which could be ambiguous otherwise: `int offset` vs. `int index`.

5. Methods

Methods are defined using a similar syntax as generics as summarized in figure 8. The following code defines a method specialization of the generic `gincr` for the class `Counter`:

```c
defmethod(void, gincr, Counter)
    self->cnt++;
endmethod
```

which in OBJECTIVE-C gives (within `@implementation`):

```c
-(id) incr {
    self->cnt++;
}
```

**Methods specializers** The *receivers* can be equivalently accessed through `selfn`\(^{11}\) whose types correspond to their class specialization *(e.g. `struct Counter*`)* and through unnamed parameters `_n` whose types are `OBJ` for \(1 \leq n \leq g\), where \(g\) is the rank of the generic. It is important to understand that `selfn` and `_n` are bound to the same object, but `selfn` provides a statically typed access which allows to treat CLOS objects like normal C structures.

**Multi-methods** Multi-methods are methods with more than one receiver and do not require special attention in CLOS. The following example defines the assign-sum operator *(i.e. `+=`)* specializations which adds 2 or 3 Counters:

\(^{11}\) `self` and `self1` are equivalent.
Class methods  Class methods are methods specialized for classes deriving from Class which includes all metaclasses:

```python
defmethod(OBJ, gaddTo, Counter, Counter)
    self->cnt += self2->cnt;
    retmethod(_1); // return self
endmethod
```

```python
defmethod(OBJ, gaddTo2, Counter, Counter, Counter)
    self->cnt += self2->cnt + self3->cnt;
    retmethod(_1); // return self
endmethod
```

**Method aliases** COS allows to specialize compatible generics with the same implementation. The following aliases define specializations for gpush, gtop and gdrop respectively:

```python
defalias(void, gput)gpush, Stack, Object);
defalias(OBJ, gget)gtop, Stack, Object);
defalias(void, gdrop)gpop, Stack, Object);
```

**Method types** In order to support fast generic delegation (section 5.2), COS must use internally the same function types (i.e. same C function signatures) for methods implementation belonging to generics of the same rank:

```python
void (*IMP1)(SEL, OBJ, void*, void*);
void (*IMP2)(SEL, OBJ, OBJ, void*, void*);
void (*IMP3)(SEL, OBJ, OBJ, OBJ, void*, void*);
...
```

The first parameter _sel is the message selector (i.e. generic’s object) used by the dispatcher, the OBJs _n are the objects used as selectors (i.e. receivers) by the dispatcher, the penultimate parameter _arg is a pointer to the structure storing the closed arguments of the generic (if any) and the last parameter _ret is a pointer to the placeholder of the returned value (if any). The responsibilities are shared as follow:

- The **generic functions** are in charge to pack the closed arguments (if any) into the structure pointed by _arg, to create the placeholder pointed by _ret for the returned value (if any), to lookup for the method specialization and to invoke its implementation (i.e.IMP_n) with the prepared arguments _sel, _n, _arg and _ret.
- The **methods** are in charge to unpack the closed arguments into local variables and to handle the returned value appropriately.

---

**Figure 8.** Syntax summary of methods.

5.1 Next method

The next_method principle borrowed from CLOS\(^{12}\) is an elegant answer to the problem of superclass(es) methods call (i.e. late binding) in the presence of multi-methods. The following sample code defines a specialization of the message gincrBy for the class MilliCounter which adds thousandths of count to the class Counter:

```python
defmethod(void, gincrBy, MilliCounter, (int)by)
    self->mcnt += by;
if (self->mcnt >= 1000) {
    defnext(void, gincr, MilliCounter);
    self->mcnt -= 1000;
    next_method(self); // call gincr(Counter)
}
endmethod
```

\(^{12}\)Namely call-next-method.
It is worth to note that next_method replaces the message sent to super in OBJECTIVE-C. By default, next_method calls the next method belonging to the same generic (e.g., gincrBy) where next means the method with the highest specialization less than the current method. But in the example above, the Counter class has no specialization for gincrBy. That is why the line 4 specifies an alternate next method path, namely gincr, to redirect the next_method call to the appropriate next method. In some cases, it might be safer to test for the existence of the next method before calling it:

```ruby
if (next_method_p) next_method(self);
```

It is worth to note that next_method transfers the returned value (if any) directly from the called next method to the method caller. Nevertheless, the returned value can still be accessed through the `retval` RETVAL.

### Methods specialization

Assuming for instance the class inheritance `A :> B :> C`, the class precedence list for the set of all pairs of specialization of `A`, `B` and `C` by decreasing order will be:

```
```

and the list of all `next_method` paths are:

```
```

The algorithm used by COS to build the class precedence list (i.e., compute methods rank) has some nice properties: it provides natural asymmetric left-to-right precedence and it is non-ambiguous, monotonic and totally ordered [35].

### Around methods

Around methods borrowed from CLOS provide an elegant mechanism to enclose the behavior of some primary method by an arbitrary number of around methods. Around methods are always more specialized than their primary method but have an undefined precedence:

```ruby
defmethod(void, gdoIt, A, A)
endmethod

defmethod(void, gdoIt, B, A)
  next_method(self1, self2); // call gdoIt(A,A)
endmethod

defmethod(void, (gdoIt), B, A) // around method
  next_method(self1, self2); // call gdoIt(B,A)
endmethod

defmethod(void, gdoIt, B, B)
  next_method(self1, self2); // call (gdoIt)(B,A)
endmethod
```

5.2 Delegation

Message forwarding is a major feature of COS which was developed from the beginning with fast generic delegation in mind as already mentioned in the previous section.

### Unrecognized message

Message dispatch performs runtime lookup to search for method specializations. If no specialization is found, the message `gunrecognizedMessage` is substituted and sent with the same arguments as the original sending, including the selector. Hence these messages can be overridden to support the delegation or some adaptive behaviors. The default behavior of `gunrecognizedMessage` is to throw the exception `ExBadMessage`.

### Forwarding message

Message forwarding has been borrowed from OBJECTIVE-C and extended to multi-methods. The sample code below shows a common usage of message forwarding to protect objects against invalid messages:

```ruby
defmethod(void, gunrecognizedMessage1, MyProxy)
  if (gunnderstandMessage1(self->obj), _sel)==True
    forward_message(self->obj);
  // delegate
endmethod
```

which can be translated line-by-line into OBJECTIVE-C by:

```ruby
- (retval_t) forward:(SEL)sel : (arglist_t)args {
  if ([self->obj respondsTo: sel] == YES)
    return [self->obj performv:sel : args];
}
```

Here, `forward_message` propagates all the arguments, including the hidden parameters `_sel`, `_arg` and `_ret`, to a different receiver. As for `next_method`, `forward_message` transfers the returned value directly to the method caller and can be accessed through `retval` in the same way.

### Fast delegation

Since all methods belonging to generics with equal rank have the same C function signature and fall into the same lookup cache, it is safe to cache the message `gunrecognizedMessage` in place of the unrecognized message. Hence, the next sending of the latter will result in a cache hit.

This substitution allows the delegation to be as fast as message dispatch, seemingly a unique feature.

### Intercession of forwarded messages

Since the closed arguments of the generic’s `param-list` are managed by a C structure, it is possible to access each argument separately. In order to do this, COS provides introspective information on generics (i.e., metadata on types and signatures) which allows to identify and retrieve the arguments and the returned value efficiently. But this kind of needs should be exceptional and is beyond the scope of this paper.

5.3 Contracts

To quote Bertrand Meyer [31], the key concept of Design by Contract is “viewing the relationship between a class and
assert (invariant(old)

the execution of the
BODY
Draft research paper for OOPSLA'09
11
summarized in figure 9, we can rewrite the method
gincr
Testing [42] (Unit developer point of view) is the complementary tool of
interfaces.
better identify the responsibilities and improve the design of
the mutual confidence between developers and users, help to
ExBadAssert
postconditions
Design by Contract provide two types of statements to express the
obligations of the caller and the callee: preconditions and
postconditions. But [32] demonstrates that EIFFEL-style
contracts may introduce behavioral inconsistencies with inheritance, thus Cos prefers to treat both pre and post
conditions as conjunctions. This is also the only known solution compatible with multi-methods where subtyping is
superseded by the class precedence list.
Contracts and inheritance In the design of EIFFEL, Bertrand Meyer recommends to evaluate inherited contracts as
disjunction of the preconditions and as a conjunction of the postconditions. But [32] demonstrates that EIFFEL-style
contracts may introduce behavioral inconsistencies with inheritance, thus Cos prefers to treat both pre and post
conditions as conjunctions. This is also the only known solution compatible with multi-methods where subtyping is
superseded by the class precedence list.
Contracts levels The level of contracts can be set by defining the macro COS_CONTRACT to one of the levels:
• NO disable contracts (not recommended).
• COS_CONTRACT_PRE enables PRE sections. This is the recommended level for production phases (default level).
• COS_CONTRACT_POST enables PRE and POST sections. This is the usual level during the development phases.
• COS_CONTRACT_ALL enables PRE and POST sections as well as test_invariant statements. This is the highest level
usually set during debugging phases.

Figure 9. Syntax summary of contracts.

its clients as a formal agreement, expressing each party’s rights
and obligations”. Most languages that support Design
by Contract provide two types of statements to express the
obligations of the caller and the callee: preconditions and
postconditions. The caller must meet all preconditions of
the message sent, and the callee (the method) must meet its
own postconditions — the failure of either party leads to a
bug in the software. In that way, Design by Contract (i.e.
developer point of view) is the complementary tool of Unit
Testing [42] (i.e. user point of view) and they both enhance the
mutual confidence between developers and users, help to
better identify the responsibilities and improve the design of
interfaces.

To illustrate how contracts work in Cos with the syntax
summarized in figure 9, we can rewrite the method gincr:
defmethod(void, gincrBy, MilliCounter, (int)by)

PRE test_assert(by >= 0 && by < 1000,
“millicount out of range”);

BODY // same code as before
endmethod

The PRE statement ensures that the incoming by is within the
expected range and the next_method call in the BODY
statement ensures that the contract of gincr is also fulfilled.

Assertions and tests In order to ease the writing of contracts and unit tests, Cos provides two standard tests:
• test_assert(expr[, str][, func, file, line]) is a replacement for the standard assert and raises an ExBadAssert
exception on failure. The (optional) parameters str, func, file and line are transferred to THROW for debugging.
• test_invariant(obj[, func, file, line]) checks for the class invariants of objects. It can only be used inside
methods and is automatically invoked on each receiver if the invariant contract level is active. The (optional) parameters func, file and line are transferred to ginvariant.

Class invariants The test_invariant assertion relies on the message ginvariant which must be specialized for
MilliCounter to be effective in the previous example:
defmethod(OBJ, ginvariant, MilliCounter, (STR)func, (STR)file, (INT)line)

next_method(self); // check Counter invariant
int mcnt = self->mcnt;
test_assert(mcnt >= 0 && mcnt < 1000,
“millicount out of range”, func, file, line);
endmethod

Here, test_assert propagates the location of the calling
test_invariant to improve bug tracking.

The POST statement test_assert checks for counter
overflow after the execution of the BODY statement and throws
an ExBadAssert exception on failure, breaking the contract.
The variable old_val initialized in the PRE statement before
the execution of the BODY statement, plays the same role as
the old feature in EIFFEL. As well gincrBy can be improved:

defmethod(void, gincrBy, MilliCounter, (int)by)

PRE test_assert(by >= 0 && by < 1000,
“millicount out of range”);

BODY // same code as before
endmethod

The PRE statement ensures that the incoming by is within the
expected range and the next_method call in the BODY
statement ensures that the contract of gincr is also fulfilled.

Assertions and tests In order to ease the writing of contracts and unit tests, Cos provides two standard tests:
• test_assert(expr[, str][, func, file, line]) is a replacement for the standard assert and raises an ExBadAssert
exception on failure. The (optional) parameters str, func, file and line are transferred to THROW for debugging.
• test_invariant(obj[, func, file, line]) checks for the class invariants of objects. It can only be used inside
methods and is automatically invoked on each receiver if the invariant contract level is active. The (optional) parameters func, file and line are transferred to ginvariant.

Class invariants The test_invariant assertion relies on the message ginvariant which must be specialized for
MilliCounter to be effective in the previous example:
defmethod(OBJ, ginvariant, MilliCounter, (STR)func, (STR)file, (INT)line)

next_method(self); // check Counter invariant
int mcnt = self->mcnt;
test_assert(mcnt >= 0 && mcnt < 1000,
“millicount out of range”, func, file, line);
endmethod

Here, test_assert propagates the location of the calling
test_invariant to improve bug tracking.

Contracts and inheritance In the design of EIFFEL, Bertrand Meyer recommends to evaluate inherited contracts as a
disjunction of the preconditions and as a conjunction of the postconditions. But [32] demonstrates that EIFFEL-style
contracts may introduce behavioral inconsistencies with inheritance, thus Cos prefers to treat both pre and post
conditions as conjunctions. This is also the only known solution compatible with multi-methods where subtyping is
superseded by the class precedence list.
Contracts levels The level of contracts can be set by defining the macro COS_CONTRACT to one of the levels:
• NO disable contracts (not recommended).
• COS_CONTRACT_PRE enables PRE sections. This is the recommended level for production phases (default level).
• COS_CONTRACT_POST enables PRE and POST sections. This is the usual level during the development phases.
• COS_CONTRACT_ALL enables PRE and POST sections as well as test_invariant statements. This is the highest level
usually set during debugging phases.
5.4 Properties

Property declaration is a useful programming concept which allows, amongst others, to manage the access of object attributes, to use objects as associative arrays or to make objects persistent. Figure 10 summarizes the syntax of properties in COS which are just syntactic sugar on top of the definition of class-objects and the specialization of the accessors ggetAt and gputAt already mentioned in section 4.3.

**Property definition** Properties in COS are defined conventionally with lowercase names:

```
defproperty( name );
defproperty( size );
defproperty( class );
defproperty( value );
```

For example, the last property definition is equivalent to:

```
defclass( P_value, Property )
endclass
```

Most notably, properties are class-objects deriving from the class Property (fig. 3) with lowercase names prefixed by P_.

**Class properties** Once properties have been defined, it is possible to define some class-properties:

```
defproperty(Counter, (cnt)value, int2OBJ, gint);
defproperty(Counter, ()class, gclass);
```

with:

```
OBJ int2OBJ(int val) { // cannot be a method
    return gautoDelete(aInt(val));
}
```

The value property is associated with the cnt attribute with read-write semantic and uses user-defined boxing (int2OBJ) and unboxing (gint). The class property is associated with the entire object (omitted attribute) with read-only semantic and uses the inherited message gclass to retrieve it.

Sometimes the abstraction or the complexity of the properties require handwritten methods. For instance:

```
defmethod(OBJ, ggetAt, Person, mP_name)
    retmethod(gcat(self->fstname, self->lstname));
endmethod
```

is equivalent to, assuming gname(Person) is doing the gcat:

```
defproperty(Person, ()name, gname);
```

**Using properties** The example below displays the name property of an object (or raise the exception ExBadMessage):

```
void print_name(OBJ obj) {
    useproperty(name);
    gprint(ggetAt(obj, name));
}
```

6. Exceptions

Exceptions are non-local errors which ease the writing of interfaces since they allow to solve the problems where the solutions exist. To state it differently, if an exceptional condition is detected, the callee needs to return an error and let the caller take over. Applying recursively this behavior requires a lot of boilerplate code on the callers side to check returned status. Exceptions let the callers choose to either ignore thrown errors or to catch them and take over.

Implementing an exception mechanism in C on top of the standard setjmp and longjmp is not new. But it is uncommon to see a framework written in C which provides the full try-catch-finally statements (figure 11) with the same semantic as in other object-oriented programming languages (e.g. JAVA, C#). The CATCH declaration relies on the message gisKindOf to identify the thrown exception, what implies that the order of CATCH definitions matters, as usual.

The sample program hereafter gives an overview of exceptions in COS:

```
1  int main(void) {
2      useClass(String, ExBadAssert, mExBadAlloc);
```
The code above shows some typical usages:

- Line 15 catches the class `ExBadAlloc` which is thrown when a memory allocation failure occurs. Throwing an instance of the class in such context would not be safe.
- Line 23 destroys the two strings whatever happened. Their initial states have been set to be neutral for these operations in case of failure.

`COS` allows to throw any kind of object but it provides also a hierarchy of exceptions deriving from `Exception`: `ExBadAlloc`, `ExBadArity`, `ExBadAssert`, `ExBadCast`, `ExBadDomain`, `ExBadFormat`, `ExBadMessage`, `ExBadProperty`, `ExBadRange`, `ExBadSize`, `ExBadType`, `ExBadValue`, `ExNotFound`, `ExNotSupported`, `ExErrno`, and `ExSignal`. Among these exceptions, `ExErrno` and `ExSignal` are special cases used respectively to convert standard errors (i.e. `test_errno()`) and registered signals into exceptions.

### 7. Performance

In order to evaluate the efficiency of `COS`, small test suites\(^{13}\) have been written to stress the message dispatcher in various conditions. The test results summarized in table 1 and figure 12 have been performed on an Intel DualCore\(^{TM}\) T9300 CPU 2.5 Ghz with Linux Ubuntu 64-bit and the compiler GCC 4.3 to compile the tests written in the three languages. The timings have been measured with `clock()` and averaged over 10 loops of \(2 \times 10^6\) iterations each. The `Param.` column indicates the number of parameters of the message split by `selectors` (open types) and `arguments` (closed types). The other columns represent the performances in million of invocations sustained per second for respectively `C++` virtual member functions, `OBJECTIVE-C` messages and `COS` messages. The tests stress the dispatcher with messages already described in this paper: `incr` increments a counter, `incrBy\((2\ldots5)\)` opt accept from 1 to 5 extra `closed` parameters (to stress the construction of `_arg`) and `addTo\((2\ldots4)\)` opt add from 2 to 5 Counters together (to stress multiple dispatch). Multiple dispatch has been implemented with the Visitor design pattern in `C++` and `OBJECTIVE-C`.

Concerning the performance of single dispatch, `COS` shows a good efficiency since it runs in average at about the same speed as `C++` and about \(\times 1.6\) faster than `OBJECTIVE-C`. On one hand, `COS` efficiency decreases faster than `C++` efficiency because it passes more hidden arguments (i.e. `_sel` and `_arg`) and uses more registers to compute the dispatch. On the other hand, `C++` shows some difficulties to manage efficiently multiple inheritance of abstract classes. Concerning the performance of multiple dispatch, `COS` outperforms `C++` and `OBJECTIVE-C` by factors \(\times 1.9\) and \(\times 5.3\). Concerning the performance of message forwarding, we have seen that by design, it runs at the full speed of message dispatch in `COS`. Rough measurements of `OBJECTIVE-C` message forwarding (linear lookup) shows that `COS` performs from \(\times 50\) to \(\times 100\) faster, depending on the tested classes.

\(^{13}\) These testsuites can be browsed on sf.net in the module `CosBase/tests`.
### Multi-threading
The same performance tests have been run with POSIX multi-threads enabled. When the Thread-Local-Storage mechanism is available (Linux), no significant impact on performance has been observed (<1%). When the architecture supports only POSIX Thread-Specific-Key (Mac OS X), the performance is lowered by a factor ×1.6 and becomes clearly the bottleneck of the dispatcher.

### Object creation
Like other languages with semantic by reference, COS loads heavily the C memory allocator (e.g., malloc) which is not very fast. If the allocator is identified as the bottleneck, it can be replaced with optimized pools by overriding malloc or by faster external allocators (e.g., Google tcmalloc). COS also takes care of automatic objects which can be used to speed up the creation of local objects.

### Other aspects
Other features of COS do not involve such heavy machinery as in message dispatch or object creation. Thereby, they all run at full speed of C. Contracts run at the speed of the user tests since the execution path is known at compile time and flattened by the compiler optimizer. Empty try-blocks run at the speed of setjmp which is a well known bottleneck. Finally next_method runs at 70% of the speed of an indirect function call (i.e., late binding) because it also has to pack the closed arguments into the generic’s _arg structure.

### 8. Component-Oriented Design Patterns
This overview of COS shows that the principles stated in the introduction are already well fulfilled. So far:

- The **simplicity** can be assumed from the fact that the entire language can be described within few pages, including the grammar, some implementation details and few examples and comparisons with other languages.
- The **extensibility** comes from the nature of the object model which allows to extend (methods bound to generics), wrap (around methods) or rename (method aliases) behaviors with a simple user-friendly syntax. Besides, encapsulation, polymorphism, low coupling (messages) and contracts are also assets for this principle.
- The **reusability** comes from the key concepts of COS which enhance generic design: polymorphism, collaboration (multi-methods) and composition (delegation).
- The **efficiency** measurement shows that key concepts perform well compared to other mainstream languages.
- The **portability** comes from its nature: a C89 library.

It is widely acknowledged that dynamic programming languages simplify significantly the implementation of classical design patterns [28] when they don’t supersede them by more powerful dynamic patterns [7, 33, 34]. This section focuses on how to use COS features to simplify design patterns or to turn them into reusable components, where the definition of **componentization** is borrowed from [16, 17]:

\[
\text{Component Orientation} = \text{Encapsulation} + \text{Polymorphism} + \text{Late Binding} + \text{Multi-Dispatch} + \text{Delegation}.\]

#### 8.1 Simple Patterns
**Creationical Patterns** It is a well known fact that these patterns vanish in languages supporting generic types and introspection. We have already seen _gnew_ (p. 5), here is more:

```c
OBJ gnewWithStr(OBJ cls, STR str) {
    return ginitWithStr(galloc(cls), str);
}

OBJ gclone(OBJ obj) {
    return ginitWith(galloc(gclass(obj)), obj);
}
```

The Builder pattern is a nice application of property meta-classes to turn it into the so called Class Cluster pattern:

```c
defmethod(OBJ, galloc, pmString)
retmethod(_1); // lazy, delegate the task to initializers
endmethod
```

---

<table>
<thead>
<tr>
<th>Tests</th>
<th>Param.</th>
<th>C++</th>
<th>ObjC</th>
<th>COS</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter incr</td>
<td>1 + 0</td>
<td>176</td>
<td>122</td>
<td>218</td>
</tr>
<tr>
<td>counter incrBy</td>
<td>1 + 1</td>
<td>176</td>
<td>117</td>
<td>211</td>
</tr>
<tr>
<td>counter incrBy2</td>
<td>1 + 2</td>
<td>176</td>
<td>115</td>
<td>185</td>
</tr>
<tr>
<td>counter incrBy3</td>
<td>1 + 3</td>
<td>176</td>
<td>112</td>
<td>171</td>
</tr>
<tr>
<td>counter incrBy4</td>
<td>1 + 4</td>
<td>167</td>
<td>111</td>
<td>154</td>
</tr>
<tr>
<td>counter incrBy5</td>
<td>1 + 5</td>
<td>167</td>
<td>107</td>
<td>133</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tests</th>
<th>Param.</th>
<th>C++</th>
<th>ObjC</th>
<th>COS</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter addTo</td>
<td>2 + 0</td>
<td>90</td>
<td>40</td>
<td>150</td>
</tr>
<tr>
<td>counter addTo2</td>
<td>3 + 0</td>
<td>66</td>
<td>23</td>
<td>121</td>
</tr>
<tr>
<td>counter addTo3</td>
<td>4 + 0</td>
<td>45</td>
<td>16</td>
<td>90</td>
</tr>
<tr>
<td>counter addTo4</td>
<td>5 + 0</td>
<td>40</td>
<td>12</td>
<td>77</td>
</tr>
</tbody>
</table>

Table 1. Performances summary in 10^6 calls/second.
This example shows how to delegate the object build to the initializer which in turn allocates the appropriate object according to its arguments, an impossible task for the allocator. The allocation of the StringLiteral uses the standard allocator inherited from Object, even if it derives from the class String (thanks to property metaclass). Now, the code:

```c
OBJ str = gnewWithStr(String, "literal string");
```

will silently return an instance of StringLiteral. This is the cornerstone of Class Clusters where the front class (e.g. String) delegates to private subclasses (e.g. StringLiteral) the responsibility to build the object. It is worth to note that each pmString specialization needed to handle new subclass is provided by the subclass itself (thanks to the open class model), which makes the Builder pattern truly extensible. Most complex or multi-purpose classes of COS are designed as class clusters (e.g. Array, String, Functor, Stream).

**Garbage Collector** This exercise will show how to simplify memory management in COS with only few lines of code. We start by wrapping the default object allocator such that it always auto-releases the allocated objects:

```c
defmethod(OBJ, ginitWithStr, pmString, (STR)str)  
OBJ lit_str = galloc(StringLiteral);  
retmethod( ginitWithStr(lit_str, str) );  
endmethod
```

This example shows how to translate strings (key) into properties (noun):

```c
defmethod(OBJ, ggetAt, Object, String)  
OBJ prp = cos_property_getWithStr(self2->str);  
if (!prp) THROW(gnewWith(ExBadProperty, _, 2));  
retmethod( ggetAt(_, 1, prp) );  
endmethod
```

```c
defmethod(void, gputAt, Object, String, Object)  
OBJ prp = cos_property_getWithStr(self2->str);  
if (!prp) THROW(gnewWith(ExBadProperty, _, 2));  
gputAt(_, 1, prp, _3);  
endmethod
```

where `cos_property_getWithStr` is an optimized version of `cos_class_getWithStr` for properties from the API of COS, which also provides `cos_class_(read,write)Properties` to retrieve all the properties of a class (and its superclasses).

**Key-Value-Observing** Adding access notifications of properties is the next step after KVC, using around methods:

```c
defmethod(OBJ, ggetAt, (Person), mp_name)  
useclass(After, Before);  
OBJ context = aMthCall(_, _mth, 1, 2, _arg, _ret);  
gnotify(center, context, After);  
next_method(self1, self2); // get property  
gnotify(center, context, After);  
endmethod
```

where `_mth` is the object representing the method itself. This example assumes that observers and objects observed have already been registered to some notification center as commonly done in the Key-Value-Observing pattern.

### 8.2 Proxies and Decorators

**Proxy** Almost all proxies in COS derive from the class Proxy which handles some aspects of this kind of class:

```c
defclass(Proxy);  
OBJ obj; // delegate  
endclass
```

```c
#define gum1 gunrecognizedMessage1 // shortcut  
#define gum2 gunrecognizedMessage2 // shortcut
```

```c
defmethod(void, gum2, Proxy, Object)  
forward_message(self1->obj, _2);  
check_ret(_sel, _ret, self1);  
endmethod
```

```c
defmethod(void, gum2, Object, Proxy)  
forward_message(_, self2->obj);  
check_ret(_sel, _ret, self2);  
endmethod
```

where the small function `check_ret` takes care to return the proxy when the forwarded message returns the delegate `obj`. 

---

**Draft research paper for OOPSLA’09**

2009/5/20
Tracer  For the purpose of debugging, COS provides the simple proxy Tracer to trace the life of an object:

```ruby
defclass(Tracer, Proxy)  # usage: gnewWith(Tracer, obj);
endclass

defmethod(void, gum2, Tracer, Object)
  trace_msg2(_sel, self1->Proxy.obj, _2);
  next_method(self1, self2);  # forward message
endmethod

defmethod(void, gum2, Object, Tracer)
  trace_msg2(_sel, _1, self2->Proxy.obj);
  next_method(self1, self2);  # forward message
endmethod

// ... other rank specializations
```

where trace_msg2 prints useful information on the console.

Locker  The locker is a proxy which avoids synchronization deadlock on shared objects that can be encountered in programming languages supporting only single-dispatch [8]:

```ruby
defclass(Locker, Proxy)  # usage: gnewWith(Locker, obj);
  pthread_mutex_t mutex;
endclass

defmethod(void, gum1, Locker)  // like smart pointers
  lock(self);  // lock the mutex
  next_method(self);  // forward the message
  unlock(self);  // unlock the mutex
endmethod

defmethod(void, gum2, Locker, Locker)
  sorted_lock2(self1, self2);  // lock in order
  next_method(self1, self2);  // forward message
  sorted_unlock2(self1, self2);  // unlock in order
endmethod

// ... other rank specializations (total of 63–6 combinations)
```

For the sake of efficiency, higher ranks use sorting networks.

Multiple Inheritance  The first version of COS was natively implementing multiple inheritance using the C3 algorithm [35] to compute the class precedence list on the way of DYLAN, PYTHON and PERL6. But it was quickly considered as too complex for the end-user and incidental as far as fast generic delegation could be achieved. Indeed, multiple inheritance can be simulated by composition and delegation with an efficiency close to native support14:

```ruby
defclass(IOStream, OutStream)
  OBJ in_stream;
endclass

defmethod(void, gum1, IOStream)
  forward_message(self->in_stream);
endmethod
```

14 OBJECTIVE-C delegation is too slow to simulate multiple inheritance.

Now, messages of rank one not understood by the IDStreams (e.g. gget, gread) will be forwarded to their InStreams.

Distributed Objects  Without going into the details, we can mention that COS already implements all the key concepts required to develop a distributed object system on the model of OBJECTIVE-C and COCOA. A challenge for the future.

8.3 Closures and Expressions  COS provides the family of geval messages (equivalent to COMMON LISP function) and the class cluster Functor to support the mechanism of closures and more generally lazy expressions and high order messages. The objects representing the context of the closure (i.e. the free variables) are passed to the Functor constructor which handles partial evaluation and build expressions. The next example shows another way to create a counter in PERL using a closure:

```perl
sub counter {
  my($val) = shift;  # seed
  $cnt = sub {  # incr
    return $val++;
  }
  return $cnt;  # return the closure
}

which can be translated into COS as:

```ruby
OBJ counter(int seed) {
  OBJ fun = aFunctor(gincr, aCounter(seed));
  return gautoDelete(fun);
}

int main(void) {
  OBJ cnt = counter(0);
  for(int i=0; i<25000000; i++)
    $&$cnt();
}
```

The line 2 creates the closure using the automatic constructor aFunctor which takes the generic function gincr and deduces its arity (here 1) from the remaining parameters, namely the seed boxed in the counter. Line 3, the message gautoDelete extends the lifespan of both the functor and the counter to the dynamic scope. As one can see, COS achieves the same task as PERL with about the same amount of code but runs more than ×15 faster. The example below:

```ruby
fun = gaddTo(x,y)  // works only with messages
fun = aFunctor(gaddTo, Var,y)  // works with functions
```

create a closure with arity 2 where Var specifies an argument placeholder and y an argument object. The message geval1(fun,x) is then equivalent to gaddTo(x,y).

The following example shows a more advanced example involving lazy expressions and indexed placeholders:
High Order Messages  The principle behind HOMs is to construct on-the-fly an expression from their arguments composition [37] — a technique known for more than a decade in C++ meta-programming. Once the expression is completed, the last built object evaluates the expression and returns the result. While C++ meta-expressions rely strongly on static types (i.e. traits) and templates to build and simplify the expressions, HOMs rely on the delegation mechanism:

• With fast generic delegation, no need to cache the message in the HOM objects as in the aforementioned paper.
• With multi-methods, no need to provide multiple HOMs for similar tasks (i.e. gfilter, gselect and gcollect).
• With lazy expression, no need to construct complex meta-expressions or to reorder compositions.

HOMs are an important tool for modern framework design since they play the role of weavers of cross-cutting concerns otherwise solved by foreign technologies based on subject-oriented [38] and aspect-oriented programming [39]. Likewise, the availability of HOMs simplify drastically the implementation of interpreters reflecting the classes and methods of the underlying language like in F-SCRIPT [43].

9. Conclusion

Some frameworks, with the help of external preprocessors or compilers, propose extensions to the C programming language, but none of them provide a set of features as consistent, complete and efficient as COS. Besides, even if the features of COS are not new, I am not aware of a single programming language which provides the same set of features with the same simplicity, portability and efficiency. Finally, very few programming languages target all the principles stated in the introduction. In particular, modern type systems which try to improve code flexibility and reusability to ease the design of generic components, tend to be lazy (static duck typing), overcomplicated (C++ ADL) and counter productive for average or occasional developers.

9.1 Related work

Ooc  This old framework uses the ooc preprocessor written in AWK [40] to provide a basic object-oriented layer. It relies on void pointers and requires much defensive programming to ensure correct use of objects. Besides, it gives full control over inheritance like in prototype-based languages.

Dynace  This framework includes the dpp preprocessor and a large library of classes [41]. Dynace features are equivalent to those of OBJECTIVE-C except that it supports multiple inheritance. However, Dynace message dispatch is about ×3 slower than COS even with jumpTo assembly code enabled and accessing object attributes is a bit awkward and relies on fancy macros (e.g. accessIVs, GetIVs, ivPtr, ivType).

Gnome Objects  The Gnome/Gtk+ Object System provides basic object-oriented programming and requires to write unsafe and unchecked code. Despite that this system cumulates all the problems of Ooc and Dynace, it is nonetheless simple, portable and implements a prototype-based object model.

9.2 Summary

COS seems to be unique by the set of features it provides to the C programming language without requiring a third party preprocessor or compiler nor any platform or compiler specific feature. The library approach allowed to explore rapidly some object models and to select the most appropriate one fulfilling the best the aimed principles: simplicity, extensibility, reusability, efficiency and portability. Moreover, the list of features is complete and consistent: augmented syntax to support object-oriented programming, uniform object model with extended metaclass hierarchy, multi-methods, fast generic delegation, design by contract, properties and key-value coding, exceptions, ownership and closures. COS features have been optimized from the design point of view, but for the sake of simplicity and portability, code tuning has never been performed and lets some room for future improvement. The 8000 lines of source code of COS can be downloaded from sourceforge.net under the LGPL license.

References


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


