

MARCO: Object Architecture Method

MARCO is an object architecture method. It suggests a new way of organizing the different models concerning systems and objects during the architectural, analysis, and design phases. Its purpose is to render a global view about the information systems and its components. This has to contribute (at least in principle) to the success of applications' portability and/or reuse of components.

The fundamentals of MARCO are:

- A parallel between systems and objects: Each system may be viewed as an object. Each object may be viewed as a system. At a certain analysis level the internal structure of an object may be ignored. Some other times an object may be considered as a complex structure of other objects.
- Service: Each system (each object) asks for services or renders services to other objects.

A functional domain = a system = an object

The development of a system analysis goes along three axes, each concerning a view of the system portability:

- more details about the process—semantic portability
- more details about the working environment—application portability
- more details about the user presentation—presentation portability

OBJECTS AND SYSTEMS

An object is an entity characterized by the following features:

1. A frontier: The object is clearly separated from its environment. This frontier is impenetrable, with the exemption of "service points" areas.
2. Services: The purpose of the object is to execute services. The possible interactions are:
 - Services are provided by the object at the request of an other object. The object responds with a rendered service, one of a set of multiple possibilities.

This article is derived from a quite extensive study conducted in the last two years by the author. MARCO may be used as a method supporting process innovation.



RETOUR

- The object may ask for services at an external object. The rendered service may belong to a set of different possibilities. This notion of service generalizes that of message.

3. Service points: The object is capable of interacting with the exterior through service points, a communication gate clustering the related services. A service point may be of server or client type.
4. A memory: The object is able to preserve the "souvenir" of past activities in attributes. An attribute preserves one of various states.

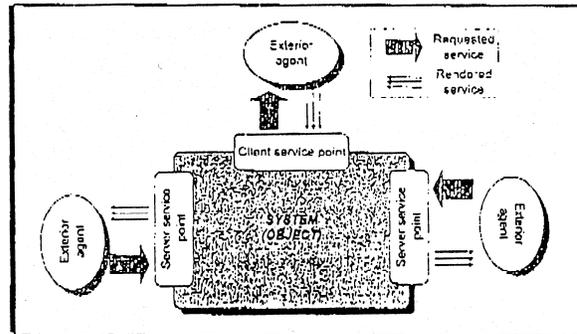


Figure 1. Object.

5. A structure: An object may contain other embedded objects. The object (Fig. 1) is accessible exclusively through service points. One can imagine that future (object) operating systems could exploit this notion directly by providing each object with a protective cover forbidding any access through areas other than the service points.

An object is defined by various models (see Table 1 and Fig. 2). An object presents at the beginning an external architecture. It is the vision that the world has about the object. The essential part of this architecture is the presentation of the object services. With this vision only, one can explore and decide upon the use or reuse of the object. Then an object, if it is a complex one (and if it is interesting to see how it is built), presents an internal architecture. The embedded objects

MARCO: OBJECT ARCHITECTURE METHOD

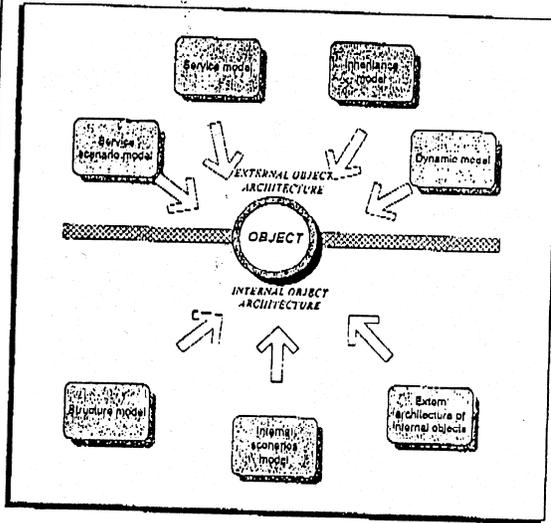


Figure 2. Object models.

are analyzed first only from their external architecture point of view. In my view this completes a "level analysis."

Model	Purpose
Services model	Presents services and service points
Service scenario model	Presents how the services are done, viewed from outside
Inheritance model	Presents the inheritance relationship of the object
Dynamic model	Presents the state change due to external events
Structure model	Presents the object structure
Internal scenario model	Presents the internal details of the scenario execution

EXTERNAL SYSTEM (OBJECT) ARCHITECTURE

An object is first perceived by the views rendered by its external architecture. The models contained in the external architecture are

- service model
- service scenario model
- dynamic model
- inheritance model

Service Model

The service model is the most important one for defining an object in MARCO. An object may receive a *service request* from an outside object. To this requested service it responds with a *rendered service*. The rendered service belongs to a set of possible responses. The object may also request a service from an external object and get in return the rendered service returned by the called object.

The requested and rendered services are clustered in *service points* (Fig. 3), the communication gate of the object. The service points may be of *client* type (requests sent only) or *server* type (requests received

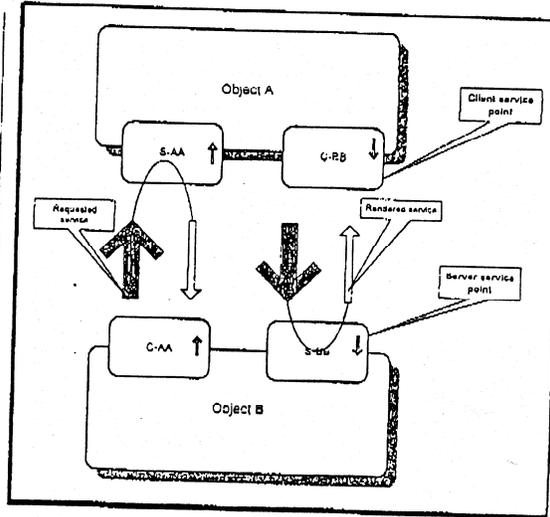


Figure 3. Service points.

only). The coupling between two objects implies the coupling between the respective client/server service points.

This model defines the service points, the services, the attributes, and the status of the object as a whole. A service request does not always imply a rendered service (e.g., a service request may indicate an internal state change, without any returned service). See the section on the service scenario model for how to choose a possibility when a rendered service belongs to a set of possibilities.

Service Scenario Model

A service scenario depicts the flow of the object's activities. A scenario is triggered by a service request and it ends with the rendered service. The initiator of a scenario may be an external object or the studied object (when it issues service requests to exterior objects). In each case a scenario follows the pattern of the service model:

- Scenario without rendered service
- Scenario with a rendered service: The choice of the rendered services depends on internal states, on services rendered by requests addressed to external objects, or on internal processes.

Dynamic Model

An object preserves the memory of its past actions. This memory stores a state of the object, one between various alternatives. The state analysis is part of the behavior analysis of the object. Two techniques may be used to represent this model: state diagram and state machines. The transitions between the states are triggered by events, which are inbound service requests or inbound services rendered to previous outbound service requests.

The service scenario model and the dynamic model describe the behavior of the object.

MARCO: OBJECT ARCHITECTURE METHOD

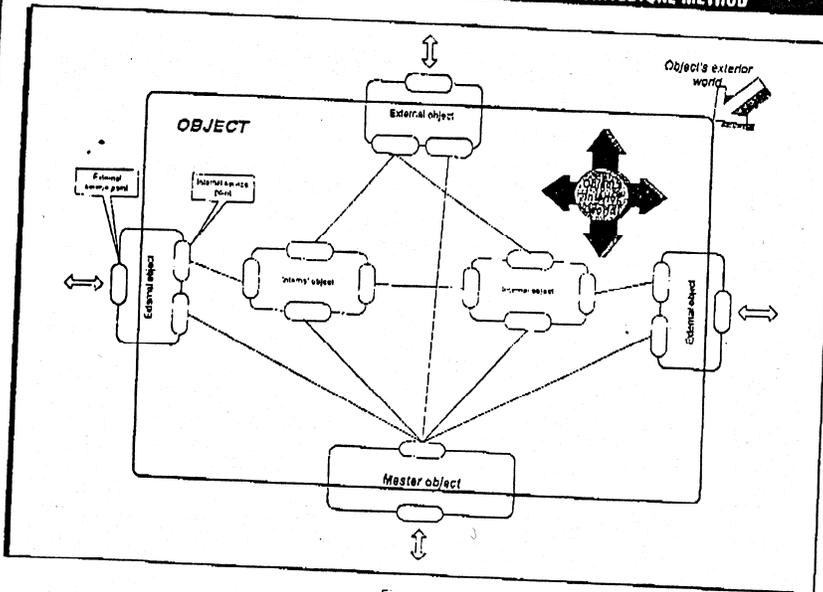


Figure 4.

Inheritance Model

This model describes the object origins. The actual object is derived from some ancestor. From the outside, the inheritance means adding new service points with new service requests and rendered services for the new object. From an internal point of view, the inheritance means new internal objects with the respective service points.

INTERNAL SYSTEM (OBJECT) ARCHITECTURE

The model contained in the internal architecture are

- structure model
- internal scenario model
- external architecture of each system component

Structure Model

This model studies the internal object structure. It identifies the component objects. The internal structure may comprise a bunch of objects. Depending on the size of the object, the internal objects may be subsystems, functional domains, or elementary objects. The object structure exhibits three objects categories (see Fig. 4):

1. Exterior (apparent) objects: These objects gather the external vision of the object (the service points). They are also provided with internal service points, used to interconnect themselves internally.

This organization corresponds to the conception that an object is more than its component parts. A special internal object gives the exclusive flavor of the system and discerns it from the trivial sum of internal and exterior objects.

2. Internal objects: These are related to the exterior objects but are invisible from the exterior.

3. A master object: This object gives a unique sense to the set of internal and exterior objects.

MARCO recommends that the structure of each object (system) should not involve over 10 component objects. This is a choice dictated by the will to manage the complexity of the projects by creating structures easy to understand and related to the application domain:

- Each component object may be built at its turn by a maximum 10 component objects, and so on.
- The object's study may be delegated easily to various team groups through natural separation lines. The overall system remains coherent.
- This policy leads naturally to the isolation of objects to be reused in other projects.

The 10-objects limit is a modular criteria (an object collection is an object in this sense). If respected, this rule allows for complexity management by

maintaining the comprehension of the system. It makes easy the localization of future changes. The 10-object limit is derived from the famous "seven plus/minus two" figure representing a result of the research concerning the capacity of the human brain to grasp simultaneously various subjects.

This decomposition rule is applied also to the objects that build the starting object (system), in iterations. At the end we have nondecomposable objects. This limit corresponds to the details judged necessary to understand and to build the application. When going from architecture to analysis and then to design, some nondecomposable objects at this stage may be split in the more detailed stage, following the same procedure: external architecture—internal architecture.

Internal Scenario Model

The internal scenario model corresponds to the action flow of the component objects to fulfill a service request. In comparison with the service scenario model, the present model shows the interactions between the service points and the services of the component objects. The 10-object limit allows for clarity in this presentation.

External Architecture of Component Objects

The component objects are presented through their external architecture, mainly the service model. The internal service model is built by taking into account the service points of the component objects

ARCHITECTURE PROCEEDINGS WITH MARCO

MARCO may be used to define a system and then to refine its analysis, to synthesize a system starting with some component blocks, or both. If we define various steps in the activity of defining and implementing a system, we can see that in MARCO the phases like architecture, analysis, and design are differentiated only by the detail de-

MARCO: OBJECT ARCHITECTURE METHOD

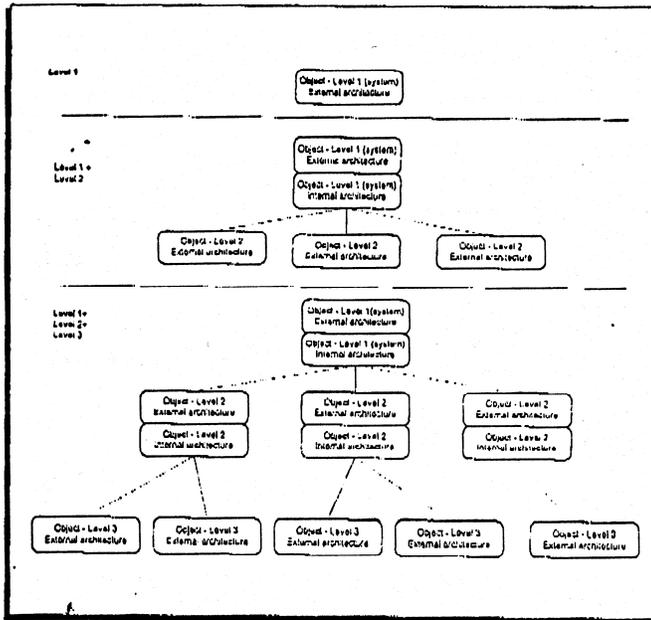


Figure 5.

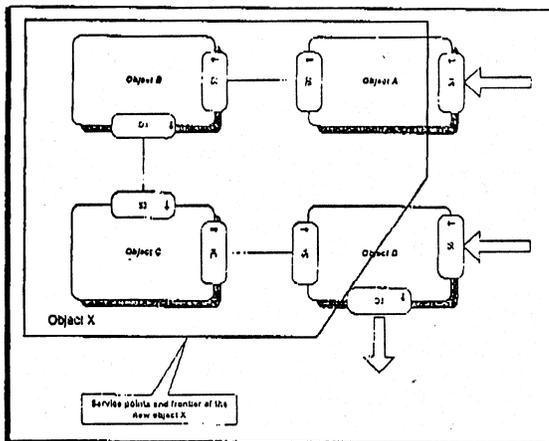


Figure 6. Object construction.

Table 2.	
First pass (first level)	Define the system (object) external architecture (level 1 object)
Second pass (second level)	Define the internal architecture of the level 1 object Define the external architecture of the level 2 objects, components of the level 1 object
...	...
nth pass (nth level)	Define the internal architecture of the nth level objects Define the external architecture of the (n+1) level objects, components of the nth level

gree of the objects. The procedure is the same: external architecture/internal architecture, with the associated model, a seamless procedure.

Generic System Analysis Process

MARCO suggests the application of the same scheme at each step in the analysis process (Table 2 and Fig. 5).

Generic System Synthesis Process

The system architecture is elaborated starting with existing objects. Each component is provided with an external architecture. If the component is complex (or interesting) it may have also an internal architecture. The synthesis starts by taking into account the external architecture of the objects constituting the building blocks of the future system.

These objects are provided with server and client service points. The connection of the objects is feasible if the service points are compatible. When various elementary objects are connected in this way, the master object will have to be defined to obtain in this way the final definition of a complex object (a subsystem). The service points of the subsystem are mainly the service points that remained unmated during the construction phase. This subsystem may be stored for future reuse in other projects. At its turn it may become a component in a higher level architecture, where it is taken into account only through its external architecture.

Organizing for Error Processing

The normality of a system is its capacity to process services in a predictive way. An abnormal situation arrives when:

- The requested service is abnormal: The service request is off-limits.
- The rendered service is abnormal: To a service request, the rendered services are incongruous (outside the expected norms).
- The rendered service is absent: To a service request there are no rendered services after a reasonable waiting time.

An error manager must be present to intervene in such cases. This manager is associated with the master object of the system and it processes the error involving the components objects. The error manager is a service point of the master object. It may use the current administrative services. It interacts with the internal objects and with the exterior objects (agents).

A system has to provide for errors processing:

- Detect the errors: This situation arrives when a component objects detects an internal abnormality. It will request a service of the error manager to inform it.
- Signal the abnormality: If a rendered service is off-limits, the object that receives it has to inform the related error manager through a service request.
- Step in to reestablish the normal use case. If it is not possible, the error manager will call upon the higher error manager.

MARCO organizes the error management by identifying an error manager for each system (complex object).

MARCO: OBJECT ARCHITECTURE METHOD

MARCO AND OTHER OBJECT-ORIENTED ANALYSIS METHODS

MARCO is essentially an organizing method using different models upon the objects and the systems, aimed at reducing the "details chaos." The MARCO models are already used in other methods, but we combine them in a new kind of organization of system analysis and synthesis:

- The same approach is used at various analysis levels, from architecture to construction, an elegant way to manage the complexity.
- The distinction between external and internal architecture provides for natural integration of subsystems (object) in more complex subsystems (objects). It allows one to take into account objects only following their external, visible presentation.
- The service points make the classification and the identification of the objects for future reuse easy. The constraints are easily recognized: compatibility between associated client and service points.
- The limitation of component object number helps the understanding and avoids a too quick atomization of the analyses (there is no forest, only a group of 10 trees, with a leader!).
- The proposed models are easy to understand. One may go up naturally, to find the global scheme of the system, or go down to see how it is done. The object system architecture is a powerful communication means of improving the dialogue between the user, the implementor, and the manager.

CONCLUSION

In this article we have presented the general lines of our MARCO method. MARCO was developed in the last two years, after we became aware of the limitations of current methods. Our customers find it quite simple and clear. It is easily understood by people at all levels: managers, implementors, and clients. MARCO is presently used in banking applications to help the definition of the next systems and to propose a seamless framework for architecture, analysis, and implementation.

Ion A. Cartian has been a consulting engineer in communications architectures for banks since 1984. A software author of various communications packages with about 1000 licenses in Europe, he is the owner of SYSOFT SA (Paris), a consulting company. He can be contacted at Sysoft, 116, Ave. des Champs Elysées, 75008 Paris, France; (v) 331 47 20 65 34; (f) 331 47 23 44 52; email: Compuserve 100060.2404.

The Booch Method, continued from page 5

may be a major functional component in certain domains (such as an electronic funds transfer system) but nonexistent in others (such as in a stand-alone computer game).

Built on top of these facilities, we find the resources that manage object storage and distribution, which collectively define the application's plumbing:

- persistent object store
- distributed object management

In traditional terms, the persistent object store constitutes an application's database, although to some degree an object store is at a slightly higher level of abstraction than a simple relational database because an object store embodies the storage properties of a database together with some of the common behavior that characterizes these objects. Distributed object management builds upon these services and lower ones in the substrate to provide abstractions for the administration of distributed as well as mobile data in a networked environment. For ex-

ample, consider a chemical engineering application that encompasses a web of computers scattered about the manufacturing floor. Some of this application's data, such as information about the inventory of various chemicals used in the process and recipes for various substances to be produced, may live on a specific node that acts as a central server for the system. Other kinds of data, such as the records about a particular production run, may be physically scattered across the network, yet appear as a logical whole to higher layers of the application.

Ultimately, the value of these two layers is that they provide the illusion to applications that objects live permanently in a large, virtual address space. In reality, an object lives on a particular processor and may or may not be truly persistent. By granting this illusion, the applications at the top of a system are ultimately much simpler, especially in geographically distributed situations. At the next highest level in the infrastructure, we find frameworks that cover domain-independent abstractions (such as various collection classes), application objects (which handle common client services such as printing and clipboard management on workstations), and the GUI facilities (which provide the primitive abstractions for building user interfaces):

- domain-independent framework
- application environment
- GUI/desktop environment

Just above this level of abstraction we find all the common abstractions that are peculiar to our given domain. Typically, these abstractions are packaged in two components:

- domain model
- domain-dependent framework

The domain model serves to capture all the classes of objects that form the vocabulary of our problem domain. For mission-critical management information systems, for example, this might include our specific abstractions of things such as customers, orders, and products, together with the business rules that apply to these things. For technical applications such as telephone switching systems, this might include things such as lines, terminals, conversations, and features, together with a specification of their behavior.

The domain-dependent framework provides all the common collaborations of these things that are specific to our domain. For example, in certain management information systems, this framework might include classes that collaborate to carry out transactions; for switching systems, this framework might provide classes that define common features such as plain old telephone service (POTS) as well as more advanced features such as call waiting, call conferencing, and caller ID.

In an earlier column, I explained how this canonical architecture adapts to real-time systems. Still, no matter what the application, it is essential to preserve the architectural integrity of a system. That goal is generally achieved by building architectures that are constructed in layers of abstraction, have a clear separation of concerns among these layers, and are simple.

All well-structured O-O architectures have clearly defined layers, with each layer providing some coherent set of services through a well-defined and controlled interface. Each layer builds upon equally well-defined and controlled facilities at lower levels of abstraction. Such architectures are ultimately simple because they reuse patterns at various levels in the system, from idioms to mechanisms to frameworks. ☐