Jean-Marc Nerson

Applying Object-Oriented Analysis and Design

Traditional analysis and design techniques imply constant paradigm shifts, since they manipulate different concepts at each different phase of software development. The object-oriented technique offers a seamless process that helps viewing the software architecture in terms of problem space elements.

This article presents an analysis and design technique relying on a set of notations and guidelines. It promotes a descriptive method that addresses both analysis and design issues. Key criteria have guided the definition of the technique: scalability, reverse engineering support, documentation aid, structuring mechanisms, systematic design support and component management support. A case study shows how the technique works and fosters the production of reusable components.

The Object-Oriented Development Process

Industrial-quality software production of today has become extremely demanding. Applications tend to be much larger and more complex and thus more difficult to develop. Their functionality is shifting from processing to system simulation and integration; from centralized to distributed computing; from text-based to graphics and multimedia-based systems [23].

Highly volatile requirements and strong competition call for even shorter development times. This conflict causes many software products to become delayed, or worse, released without adequate production quality.

The importance of application portability among a large number of rapidly changing hardware platforms and the need to be easy to learn and understand by end users with different backgrounds, make things even more difficult.

Therefore, there is no longer any time to waste on reinvention or inefficient implementation of well-known algorithms and user interface techniques.

In the long run, object-oriented software is aimed at helping to simplify the way we view the real world as it is (or as it should be) and translate our view into software systems. Object-oriented techniques exist to help manage the complexity according to some key points:

- Object-oriented architectures are decentralized;
- Classification is part of the system structure;
- The same ideas and concepts are manipulated from the requirements phase down to the implementation phase.

The object life cycle shown in Figure 1 and inspired by [9, 16, 18], conveniently reflects a development scheme in which a knowledge base represented by libraries of reusable and pluggable components impact the different production phases. The class reuse and generalization process is an iterative process influencing both analysis (reuse of frameworks [6]) and design (reuse of classifications [10]).

Compared to traditional techniques, object-oriented development is a seamless process: there is no paradigm shift between the different stages of the life cycle. Uniform principles apply throughout the development process. If types of objects identified during system analysis are specified by a name and a precise set of properties, they will translate into syntactical units (usually called "classes") in the final program. This observation is both good news and bad news.

The good news is that once the intellectual process of object-oriented development is properly understood and mastered, it can be successfully used (with some variations imposed by the levels of abstraction) to the different development stages. Tracing requirements becomes easier since manipulation of entities is more natural and smooth.

The bad news is that whenever inappropriate types of objects are selected, or whenever awkward structuring choices are made, the final architecture reflects these poor decisions.

Because of the continuous development process involved, object-oriented techniques tend to blur the borderline between analysis and design.

In the next sections, we will study how an object-oriented analysis and design method and technique is applied. The technique used is based on a method and notation called BON (Better Object Notation [14, 15]). Although the BON notation is both graphical and textual, we shall use the graphical form. In the remaining text, all BON-specific terms will appear in ***bold-italic*** type when first introduced.

Object-Oriented Analysis with an Example

The software to be developed is intended to automate the reservation and invoicing system of a car rental

company. A short outline of the major requirements is given below.

- *Vehicles are taken from one location and returned to the same location.*
- *Different models of car are grouped into a small number of price classes.*
- *Different rental plans are available, with a special weekend rate to attract nonbusiness customers.*
- *The price charged is established in advance.*
- *Free options are: automatic or manual transmission, two or four doors, smoker or nonsmoker car.*
- *Nonfitted extras are: roof rack, trailer, snow chains, child seats. These extras are charged to the client.*
- *The system must handle block booking of cars and keep track of car availability.*

Object-oriented analysis tries to identify the type of objects that map into elements of the application domain to be modeled. This activity helps to find the major relationships between the different types of objects considered as class instances. Classes are defined with the information they maintain, the services they provide, the constraints they comply with and how they relate to other classes. Analysis classes must satisfy some functional requirements and usually reflect a certain system viewpoint. At the analysis level all identified class information is considered public.

Analysis involves some key activities that represent things to be performed by the analyst to get a better understanding of what needs to be done. In that respect, BON provides a notation and a set of guidelines and recommendations applicable to the preanalysis and analysis phase down to detailed design; the result being a starting point for the final class programming in some object-oriented language.

The notation is backed with a set of guidelines that specify activities and deliverables. Although activities are often listed in sequential order, they are iterative in practice. For instance, there is sometimes no clear distinction between what really belongs to analysis and what really belongs to design. Very roughly one could say that analysis becomes design whenever implementation decisions are taken, whenever nonpublic information is introduced in a system, or whenever newly introduced classes do not relate to problem space objects.

Figure 1. The object-oriented software life cycle

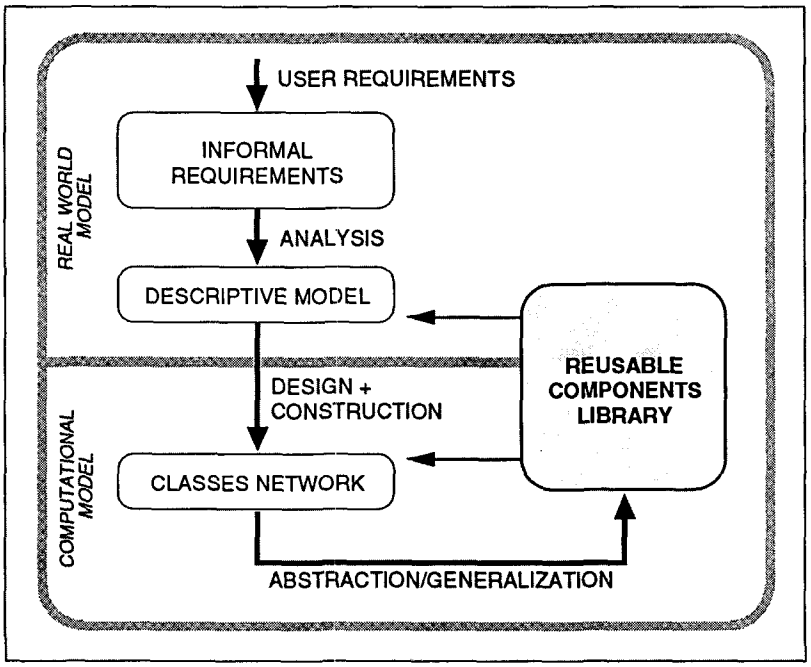


Table 1.
Cluster chart of the car rental system

CLUSTER CHART: *CAR RENTAL*	
CLASS	**DEFINITION**
CLIENT	Car renter, individual or corporate customer
CONTRACT	Rental terms with payment conditions
RENTAL	Rental information completed when taking out and returning a vehicle
VEHICLE	Automobile selected from the rental fleet
MODEL	Description of selected features
RATE	Pricing conditions

Finding, Naming, and Clustering Classes

Looking for objects and classes is the very first step of object-oriented analysis. It resembles the activity of describing a picture viewing the problem space for which the system borderline is the frame.

Both imagination and abstraction guide the analyst. The subtle effort is to decouple "nouns" potentially representing services from those mapping to effective candidate classes. No miracle formula exists in that respect besides experience and recall of good practices. Some guidelines exist, but better serve as checklists to things not to miss.

We retain basic and simple ideas as principles: the aim of analysis is

to put some order to our perception of the real world. The result is not to produce something that complicates the problem to solve or the reality to map. On the contrary, the purpose is to simplify, to master the complexity by reformulating the problem. Analysis must remove noise and overspecification, find inconsistencies, postpone implementation decisions, partition the problem space, take a certain viewpoint and document it. Object orientation simply adds structuring mechanisms for defining relationships between system elements and decentralizing local decisions.

A system usually interacts with different users. Users are not equally qualified to access specific pieces of information. There are various reasons for this: the level of responsibility in an organization; access rights (user, administrator); the level of confidence (novice, advanced user); the level of expertise or knowledge of the application domain. Occupational names give some ideas about possible classes for users: *engineer, employer, customer,* etc.

Large systems are expected to be used in completely different contexts. For a company using a system operating through a communication network *branches, affiliates, subsidiaries,* and other *manufacturing plants* may be similarly described regardless of the geographical location.

For other specific systems, the place of operation impacts the behavior. In the case of an embedded system, the place of execution may configure the system according to external constraints: severe or protected environment, ability to tune the time/space tradeoff, ability to limit or extend the accuracy of a computation depending on the context.

Since object-oriented architectures are flexible, they can model different problem space descriptions directly as part of the system description.

Information systems reflect the way organizations work. In the case of communication systems, information is presented differently according to implied actors. A military system will not display *ground information* in the same fashion for the army general as for the frontline soldier. Actors in an organization will access the same information, but it will be presented differently. The term "organization" here must be understood not only as the mapping of the work breakdown structure onto a responsibility assignment chart but also as a way information must be detailed or not, depending on the actors' levels of responsibility.

An information system behaves as a set of black boxes of which the intrinsics are hidden and the only visible part is the list of services and states provided to the user. Although it may be tempting to view the system as a collection of functions, the object approach enforces the use of data abstraction that encapsulates services. Therefore, even if services come to mind first, one should look for the underlying classes that represent the grouping of these operations.

Many systems can be defined, at the requirement levels, as state machines. A state-transition diagram, in this case, is not an implementation technique, simply a convenient way to express how the system changes according to internal events. It is generally too complicated, however, to represent the entire logic using this model. Looking at the information system modeling techniques, diagrams are produced to stress the information flows. Such diagrams are also a variation of the state-transition diagram and can lead to the production of system classes.

A first principle is to consider classes as representations of abstract data types as opposed to "things they do," which is far too close to procedural techniques. A class has an internal state and offers services.

Then, classes are grouped into ***clusters.*** The grouping factor may vary, but in principle it is simpler to designate a certain viewpoint as the main focus of interest.

Clusters play various roles. During analysis they help grouping classes according to proximity criteria based on a subsystem functionality, an abstraction level or an end-user standpoint. During design they are often used as a structuring technique to selectively visualize the coupling between classes. In any case they should not be confused with classes.

From the analysis of the car rental system, an initial set of classes comes to mind and participates in the definition of a first cluster.

In the early stages of analysis, BON uses various charts for describing the different types of objects. These charts are used as a communication tool between end users and analysts and were inspired by the CRC technique [1], but with a different rationale.

The ***cluster chart*** lists classes participating in the identified clusters and gives a short definition for each of them. It is very common to start with only one general cluster and then come up with new ones after doing some class grouping. A simplified cluster chart for the car rental system is given in Table 1.

It is also important to locate the place of a class within the overall structure. This means that finding related classes is more important than finding a single class. Design will decide whether classes are linked by derivation, association, generalization, or specialization relationships. Table 2 illustrates how clustering may group together with a class, say *AUTOMOBILE,* quite different classes depending on application domains.

We will then continue our system analysis based on two types of representation: a ***dynamic model,*** showing parts of its behavior and a ***static model*** describing its structure.

Events and Object Communication Protocols

Two different kinds of events that act on system behavior must be sorted out: ***external events*** and ***in-***

ternal events.

External events are actions initiated by the external interactors that produce an incoming data flow that crosses the boundary of the system. They refer to stimuli received from the outside upon which the system reacts according to a certain behavior. External events help to find classes that interface the system with the outside world. They should not be considered as "unexpected" events. They are things that can possibly happen but over which the system has no control on whether or when they may occur. External events may become classes or input parameter data passed to class features.

Internal events are time-related stimuli that may be relative or absolute. They refer to parts of system behavior that simply translate into class features that will lead to messages defining the ***object communication protocols.*** Since they act on the system state, they also translate into assertions that control the proper behavior of the system.

This technique relates to principles applied in "Object Behavior Analysis" [19].

Table 3 gives an example of events applicable to the car rental system and can be compared with a nonrelated system (a traffic light controller).

Events help to introduce the dynamic model that complements the static system structure. It highlights objects relevant to selected system behaviors. The dynamic model consists of scenarios demonstrating significant object communication protocols. The purpose is twofold: it helps to validate the static model and to make sure objects are reachable from others; it maps the system behavior better as opposed to the static model that only reflects the structure.

Deciding which model should be produced first really depends on the nature of the application and the analyst's level of confidence.

Defining Classes and Sketching Out the Kernel Architecture

From the initial list of classes, ***class charts*** are then filled out. For each class entry in the initial cluster chart, a class chart is defined according to three basic types of information:

- The ***questions:*** what information can other classes ask from the class?
- The ***commands:*** what services can other classes ask the class to provide?
- The ***constraints:*** what knowledge must the class maintain?

The chart is structured in three columns, filled with free-format English text.

The class chart describes a type of object that should correspond to a class entry in the cluster chart. It is also possible to state that the described type of object is suspected to behave like other types of object. This may help refining the inheritance classification later in the process. Examples of class charts are given in Figure 2.

Analysis proceeds using a more formal way to represent the system structure and behavior. The static model shows the system structure by its decomposition, its elements and their relationships, along with the constraints that the classes must fulfill. Two levels of detail are supported: the class level and the cluster level that defines relationships between logically grouped classes. The model introduces typed information very early in the analysis process. In practice, this means that any class definition introduces a type definition.

Relationships between clusters are mostly structural. They help scaling up or down the system according to grouping factors or abstraction levels. Class relationships may capture additional semantic information that translate into ***class annotations.*** Relationships between classes are represented in a quite different manner compared with

Table 2.
Class *AUTOMOBILE* with related cluster classes

APPLICATION DOMAIN	CLASSES RELATED TO *AUTOMOBILE*
Automobile manufacturer	*LIMO, COUPE, SEDAN, STATION_WAGON*
Traffic-light controller	*PEDESTRIAN, BIKE, VEHICLE*
Repair Station	*FOREIGN_CAR, DOMESTIC_CAR*
Car rental company	*TRUCK, TRAILER, COMPACT_CAR, LUXURY_CAR*

Table 3.
Event charts

APPLICATION DOMAIN	EXTERNAL EVENT	INTERNAL EVENT
Car rental company	*Customer makes a car reservation.* *A rented car breaks down.* *A new car joins the fleet.* *A car is returned.*	*Rental contract printed when car returned.* *Tank refilled on return.* *Mileage checked on return.* *Car inspection done on leave and on return.*
Traffic-light controller	*A vehicle is arriving at the intersection.* *A vehicle is leaving the intersection.* *A road is closed to traffic.*	*Light turns green.* *Red light starts flashing.*

techniques such as the entity-relationship model. Tables 4 and 5 make a parallel between the entity-to-entity relationships in the relational model and the class-to-class structuring mechanisms of the object model. The class structuring mechanism is based on two basic relationships: the ***inheritance relationship*** and the ***client-supplier relationship.*** The client-supplier relationship usually encompasses two things: the ***association relationship*** and the ***aggregation relationship,*** as explained in [13].

Classes always belong to a cluster usually displayed in abstracted form, that is only with their header. Figure 3 illustrates static and dynamic model notations used in BON.

Table 4.
Semantics of entity relationships in the relational model

TYPE OF RELATIONSHIP *(multiplicity)*
has_a*(1:1)*
owns, contains, is-contained-in*(1:m)*
consists-of*(m:m)*
is-a*(1:1)*

Object-Oriented Design with an Example

Object-oriented design transforms the analysis classes into a computerized model that belongs to the solution space. This activity starts from the analysis classes and produces additional types of objects that become classes not directly related to the problem space. These classes extend, generalize, or implement the initial set of analysis classes. Primitives introduced during design are not necessarily public and may relate to implementation decisions such as to determine which

Table 5.
Semantics of inheritance and client-supplier relationships in the object-oriented model

CLASS TYPE OF RELATIONSHIP *CLASS*			
inheritance relationships	*descendant* is-a *parent* *descendant* behaves-like *parent* *descendant* implements *parent* *descendant* combines *parents* *parent* defers-to *descendant* *parent* factors-out *descendants*	Client/Supplier relationships	*client* uses *suppliers* *client* needs *suppliers* *client* has-a *supplier* *client* consists-of *suppliers* *supplier* provides-to *clients*

CONTRACT

Cluster name: *CAR RENTAL SYSTEM*

TYPE OF OBJECT: Rental terms with payment conditions		Behaves like: *PURCHASE_ORDER*
Questions	**Commands**	**Constraints**
Means of payment. *Individual or corporate customer.* *Rental documents.*	*Invoice client.* *Acknowledge a reservation.*	*Corporate customers do not get week-end rate discounts.*

CLIENT

Cluster name: *CONTRACT ELEMENTS*

TYPE OF OBJECT: Car renter, individual or corporate customer		Behaves like: *PERSON*
Questions	**Commands**	**Constraints**
Name. *Address.* *Corporate customer.* *Special rate.*	*Enter in date base.* *Give special rate.*	*Client has already reserved or rented a car.*

VEHICLE

Cluster name: *RENTAL PROPERTIES*

TYPE OF OBJECT: Automobile selected from the rental fleet		Behaves like: *RENTED_ITEM*
Questions	**Commands**	**Constraints**
Model of car. *License_plate.* *Availability.* *Departing location.* *Returning location.*	*Check mileage.* *Refill gas tank.* *Change oil.*	*Departing and returning locations are the same.*

RENTAL

Cluster name: *CONTRACT ELEMENTS*

TYPE OF OBJECT: Rental information completed when taking out and returning vehicle		Behaves like:
Questions	**Commands**	**Constraints**
Selected automobile. *List of authorized drivers.* *Selected insurance policy.* *Applicable rate.* *Extra choices.* *Starting and ending mileage.* *Departing and returning dates.*		*Starting mileage <= ending_mileage.* *Departing date <= returning date.*

Figure 2. Class charts of the car rental system

classes will interface to the outside of the system, deal with machine or system dependent information, lead to persistent objects, be in charge of error recovery, etc.

Describing, Indexing and Instantiating Classes

During design with BON, class charts are translated into ***class descriptions.*** These class descriptions rephrase the same information in a more structured and formal manner that will help the generation of prefilled class templates in an appropriate object-oriented language. The description details class features and contracting conditions. Class descriptions first focus on visible features.

Class chart column entries become comments associated to class features; which means that any consistency checking at that level can only be done by an automated tool. The translation scheme is straightforward:

- Questions are mapped into ***attributes*** (state variables) or ***functions*** in the Eiffel sense;
- Commands are mapped into ***procedures;***
- Constraints are mapped into ***assertions*** and ***class invariants;***
- Behavior resemblance translates into inheritance (see [11]).

Routine signatures ***(input*** or ***output parameters, pre-*** and ***postconditions)*** are also listed. In addition to this, one should recall that internal events may become class features.

Figure 4 is the translation to class descriptions of the class charts defined during the analysis stage. The explanation of some symbols is also given in an associated table. Classes are now fully described: with their header and their body. The class body is decomposed into different parts. The most commonly described parts are: the reference to a direct parent, the list of typed features with their assertions and comments, if any, and the class invariant, if any.

For instance, the *RENTAL* class should make sure that whenever a vehicle is returned, it has previously been taken out. This obvious statement avoids possible misuse of the system. There is always a strange situation which is a potential source of error: drivers of the same company that have each rented a car

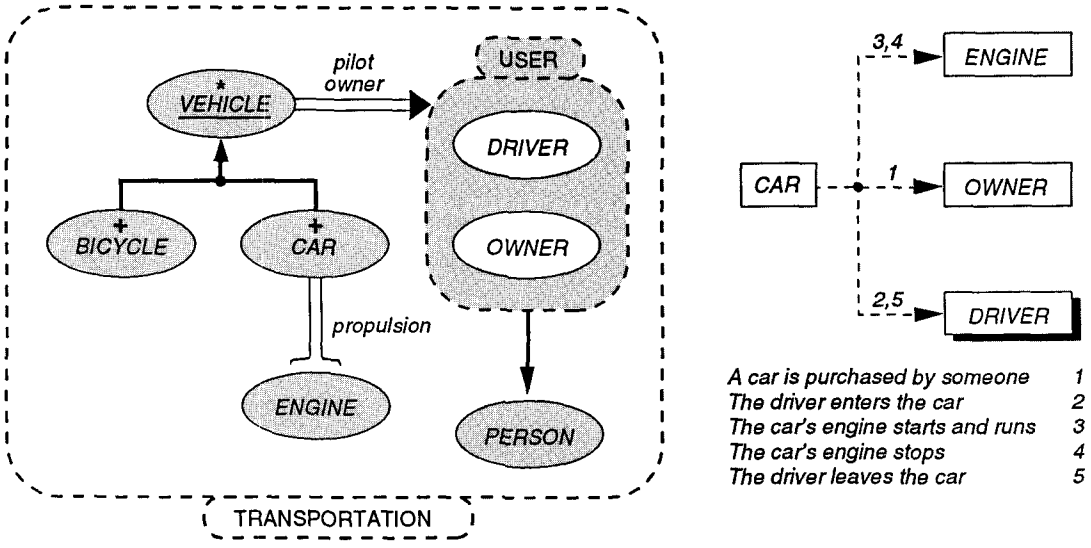


Figure 3. Graphical notations used in BON static and dynamic models

Static model (left side):

- Clusters are represented with rounded corner rectangles drawn with dashed lines and are tagged with a name.
- Classes are represented as a name inside an ellipse, with optional annotations: deferred classes are topped with a star sign, non-deferred descendant classes are topped with a plus sign, reused classes have an underlined name.
- Inheritance relationships are represented with a single line ending by an arrowhead oriented from the descendant to the parent. Client-supplier relationships are represented with a double line ending by an arrowhead in case of association and with an open curly bracket in case of aggregation. The double line may be tagged with class feature names involved in the client-supplier relationship. Relationships are defined between classes and can be extended to clusters

Dynamic model (right side):

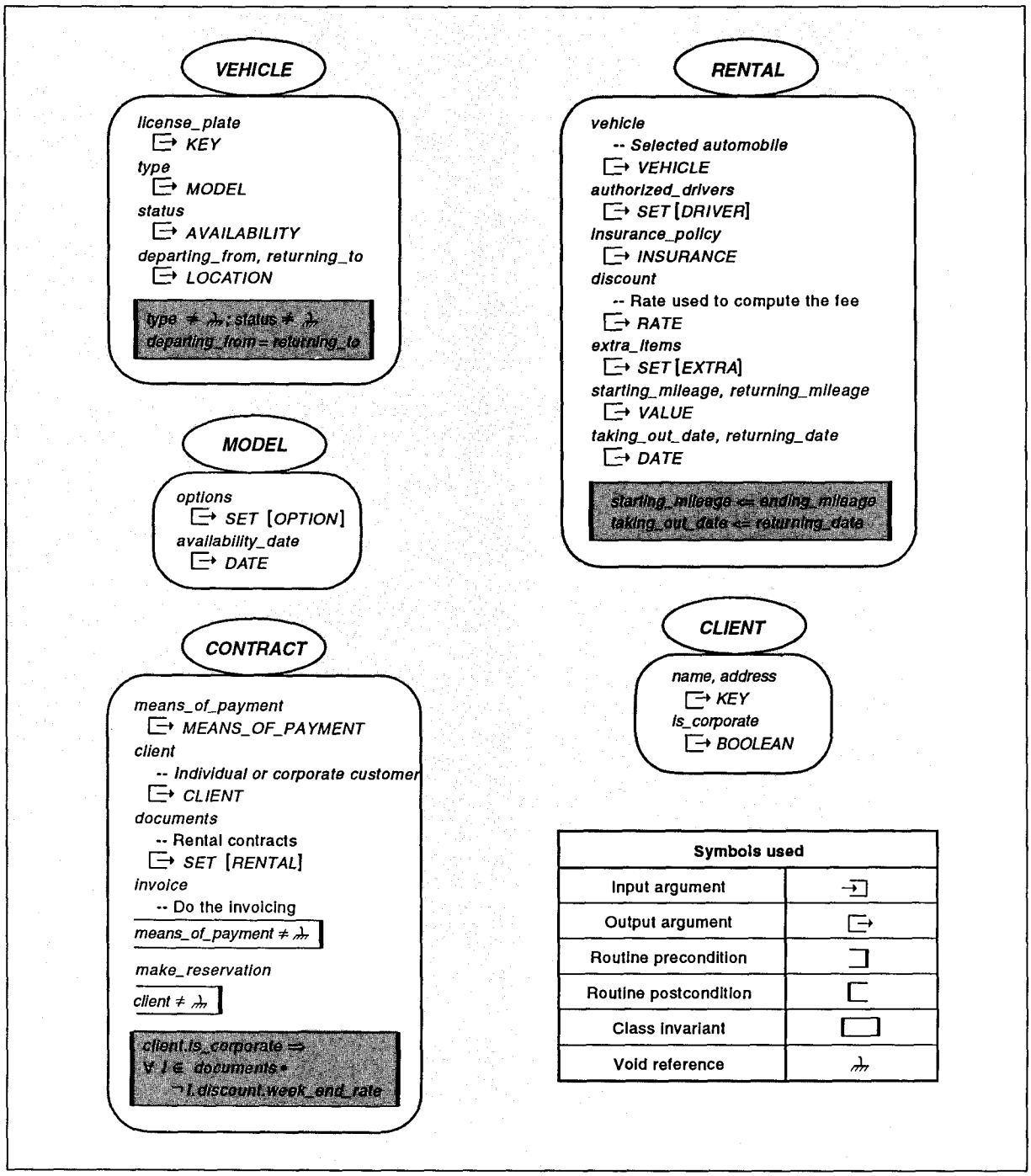
- Objects are represented inside rectangles. A shadowed rectangle denotes multiple instances.
- Communication protocols are represented with dash lines, labeled with numbers. These numbers are then referred to in commented scenarios

and then switched them without notifying the rental company. In that case, the class invariant serves as a system consistency checker.

In the *CONTRACT* class, the invariant translates a management rule stated in the initial set of requirements: corporate customers do not have access to special weekend fees. From the description of the classes, a first draft of the architecture is then defined and appears in Figure 5. A dynamic scenario is given in Figure 6. The design technique emphasizes the system flexibility with respect to its possible extensions or adaptations. High-level classes can easily be customized or particularized using inheritance.

Classes are application-dependent. The only way to solve the problem of finding suitable classes for reuse is to add class indexing

Figure 4. Class descriptions of the car rental system



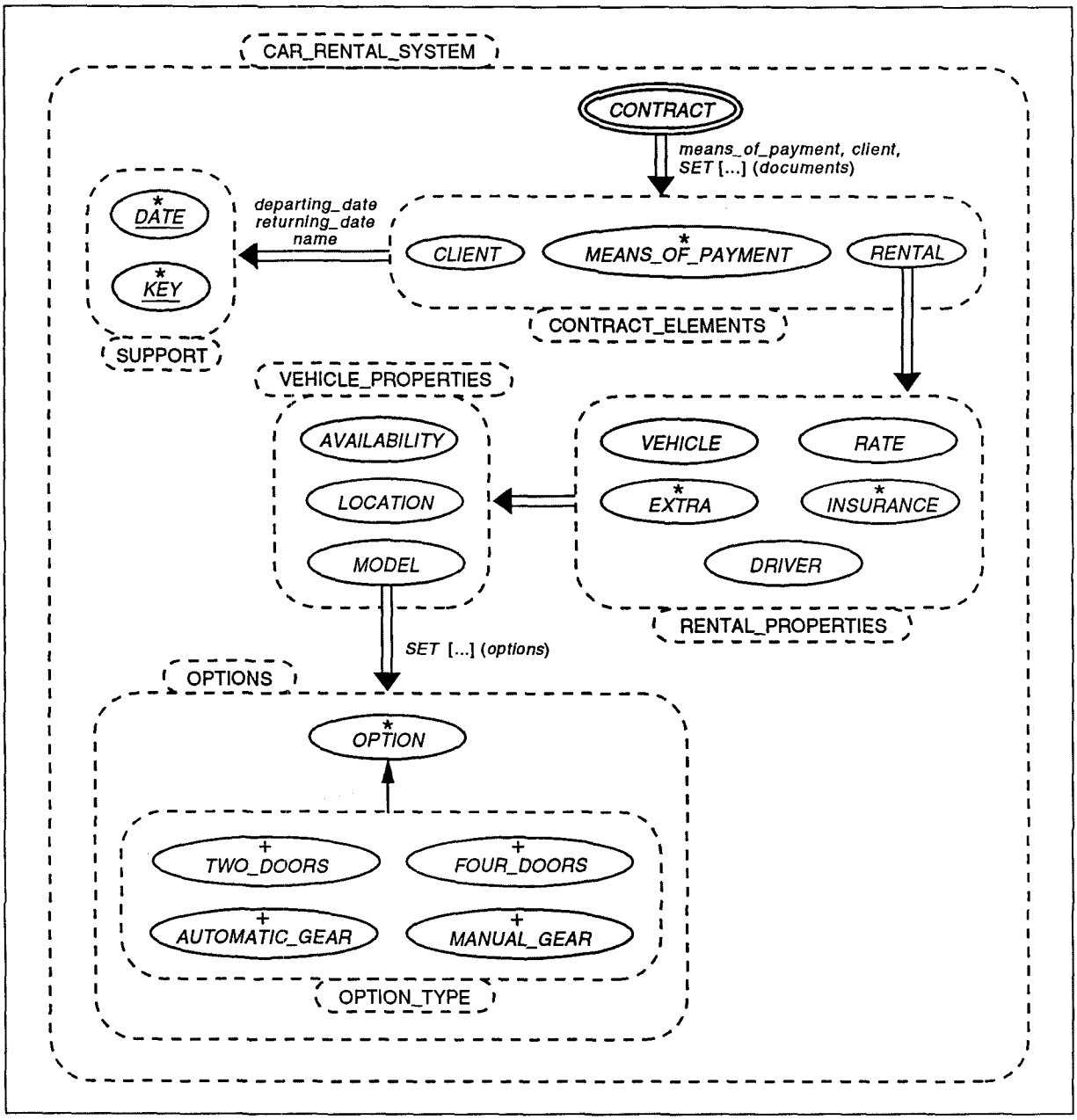
and documenting information to retrieve them easily from libraries or databases of components according to given selection criteria.

To implement automated tools managing indexing techniques, it is necessary to prepare the future reuse by enforcing design guidelines by which any completed class should be documented according to a prefilled header template. Class header templates include references to the covered application domains, to project management information, to requirement documents, etc. An example of such an indexing clause is given in Figure 7.

Once analysis and design classes are all identified, we need to know how their instances are created under certain circumstances. The static architecture only defines class relationships; the dynamic model only emphasizes selected communication protocols between objects. Therefore, a way to map one model to the other is to produce during the design phase an ***object creation chart.***

The purpose of this chart is to determine which class is responsible for the creation of the instances of other classes. At this level of description, one should concentrate only on classes found during the

Figure 5. Partial static model of the car rental system



very first stages. Table 6 shows an example of the object creation table corresponding to the car rental example reviewed earlier.

Abstracting and Classifying

What really makes object-oriented software so attractive is that its structure "is" its own classification and that this classification is achieved so as to be general enough to accommodate any possible extension or modification without impacting or complicating the existing architecture.

Figure 6. Example of a dynamic scenario

Figure 7. Indexing clause of the class *VEHICLE*

Referring to [10], which concerns the design of a library of reusable components, it is interesting to note that any of a wide range of data structures could always be linked to one of the following classification criteria:

- The *storage;* a classification dimension that captures whether the data structure is bounded, fixed, resizable, . . .
- The *traversing;* a classification dimension that captures the structural relation between the data structure elements, . . .
- The *access method;* a classification dimension that captures the ways elements are accessed.

Thus, it becomes much easier to produce any kind of data structure simply by picking and assembling classes from these three basic classifications.

Table 6.
Object creation table

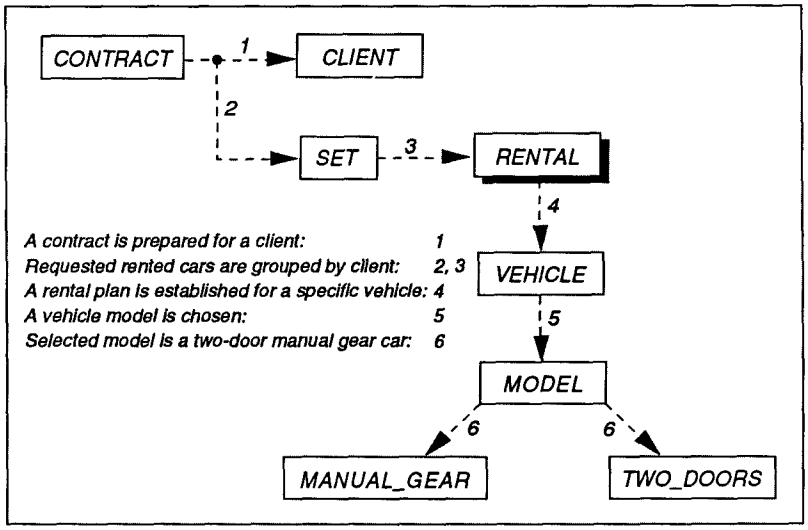
Type of object	Creates
CONTRACT	*RENTAL, RATE*
RENTAL	*VEHICLE*
VEHICLE	*MODEL*

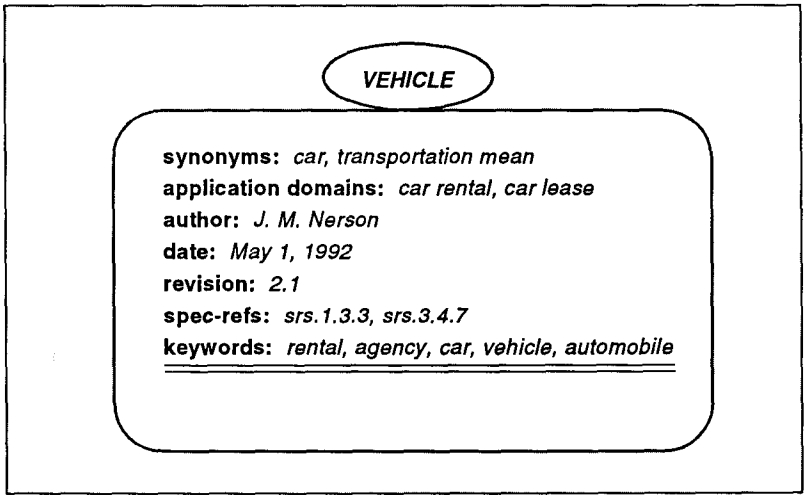
Yet a classification is never perfect. It is intended to reflect as closely as possible a reality which has no simple order. A nonsoftware-related example is the periodical classification of chemical elements by Mendeleïv: it fits reality very well, but not exactly. There are some "holes"; not because mineral elements are yet to be found, but because in some cases the classification scheme is better represented in 3D whereas it was initially designed in 2D.

Back to object-oriented software, assume that a library management software is designed around the assumption that book *authors* are human beings. To capture this idea, a classification is defined so as to make class *AUTHOR* a descendant of class *PERSON*. Unfortunately, some time later, a new book is added to the library stock that is not written by a person but by a group of unknown authors using a pseudonym. What to do then without damaging the existing architecture? The only possible solution is to define a new class, *WRITER* for instance, probably a deferred one, which is a new parent of *AUTHOR* (this will not change the interface of *AUTHOR*) and to extend the classification down from the *WRITER* class so as to introduce a class *ACRONYM_AUTHOR*.

As a conclusion we can state:

- A classification, properly laid out by the analyst according to the understanding of the problem space, models reality;
- Object-oriented facilities permit modification of the model without





breaking the core architecture when extensions or special cases show up.

Generalization of the Example

At the completion of the object-oriented analysis and design, before class coding has even started, a generalization process must begin. Looking at our system architecture, we may wonder if it is sufficiently general to maximize future reuse. The answer usually varies according to the domain of activity of the development team. In the case of a large software house, it may be appropriate to do some extra work to generalize the initial architecture. High-level class abstractions usually do not come first to mind. The improvement process often results in the following scenario:

- A first version of the system is written.
- Later, specialized versions of existing classes are written.They are heir classes of the existing classes, extending or reimplementing parent features.
- Since the initial set of classes appears too problem-specific, some common features are factored out into very high-level classes. These high-level classes serve as parent classes of the classes belonging to the initial system and to the classes newly introduced. These new classes keep the same interface as the ones they are parent of.

Before the implementation of the car rental system is completed, one may ask how to pave the way for coping with other systems similar to our initial problem such as:

- A boat rental system (motor boat and sail boat);
- A sport equipment loan system (snow skis, diving material, etc).

Referring to our initial car rental system, many common features exist, but other elements must change: there is little chance that the price of a pair of skis can be linked to a number of miles, . . . for example.

Even applications such as a hotel reservation system or a box office ticketing system have similarities with our initial system. From this observation, two options are possible:

- Do nothing and the benefit gained from applying object-oriented techniques is limited to the structuring of the application.
- Do some extra work involving the search for more general structures.

Looking at our current system architecture; classes inside the *CONTRACT_ELEMENTS* cluster can possibly be kept after some minor modifications. Classes inside the *RENTAL_PROPERTIES* cluster are much too problem-specific. To be generalized, new classes have to be introduced as ancestor classes. Classes such as *VEHICLE, DRIVER, RATE, INSURANCE* may now be defined as descendant classes of more general and abstract classes that could respectively be: *PRODUCT, CONSUMER, PRICE, WARRANTY.*

These classes introduce features that will be implemented or reimplemented in our previously defined classes. For instance, the *PRODUCT* class is initially defined by abstracting features originally introduced in the *VEHICLE* class. The car rental system classes listed inside the *RENTAL* cluster can now be rewritten according to the generalization process. Consequently, the *VEHICLE* class changes as detailed in Figure 8.

Some class invariants initially introduced in the *VEHICLE* class are now moved up into the *PRODUCT* class. Since feature *license_plate* is adapted from feature *serial_number* introduced in parent class *PRODUCT,* a plus symbol sign (+) is appended to the feature name. Features listed in the *VEHICLE* class now simply address car rental system specifics. It is worth noting that some features now introduced in class *PRODUCT,* such as *serial_number* or *stocking place,* may simply be kept under a new name in class *PRODUCT,* which is why their names are also appended with a plus sign.

This will not happen though, with feature *type* that is only relevant to the class *VEHICLE* as outlined during our initial analysis.

BON and Tools

The methodology and notation presented result from an ESPRIT Project ("Business Class" [8]) in which an object-oriented analysis and design technique was developed after experimenting on pilot projects different existing methodologies [2, 5, 7, 17, 20, 22, 24]. Specific attention was also devoted to possible enhancements of a modeling technique named O* [3, 4] with a strong database orientation and some similarities to OOSA [21].

This inspired the BON object-oriented analysis and design model and notation designed so as to support the following capabilities:

- **Analysis and design:** the formalism helps the analyst in sketching a first set of classes and relationships that can directly be translated into a system design, itself translatable into a set of programming language classes.
- **Scalability:** the formalism for representing groups of classes scales up and supports problem partitioning based on layers of abstraction using class clustering techniques.
- **Reverse engineering:** since any kind of information is stored in a coherent internal data structure, it is possible to reuse existing systems and translate them back into a schematic form. This enables the analyst to visualize existing class libraries not developed with a modeling technique, or for which the accompanying analysis and design documentation does not come with the off-the-shelf product.
- **Documentation:** any element that appears in the schematic diagram or in the textual form should be traceable. A repository of classes, properties, relationships and dependencies is maintained and can

be queried to produce browser-like information or cross-reference forms.

• **Structuring mechanism:** the model offers two graphical representations: a static diagram and a dynamic graph. The static diagram represents classes and clusters of classes all linked through different kinds of relationships. It permits the definition and visualization of well-decentralized software architectures promoted by object-oriented techniques. The dynamic graph illustrates communication scenarios between objects.

• **Systematic design:** the model fosters the application of the contract model between classes. Assertions such as pre- and postconditions of class invariants, can be expressed with a formalism relying on symbols commonly used in set theory and then directly implemented in Eiffel.

• **Component management:** software elements such as classes, features, clusters, relationships are kept under configuration management control, thanks to the indexing clauses. Each class is known by its version and other key information can potentially be interfaced with adapted querying tools.

A summary of the different methodological steps to follow is given in Table 7. Any object-oriented analysis and design technique must be backed with supporting CASE tools. In the scope of "Business Class" a workbench supporting BON is being implemented: **EiffelCase.** This workbench, designed with BON and programmed in Eiffel [11], consists of two different components.

The first EiffelCase component is a drawing tool that supports the notation and generates class templates from an internal form of clusters and system dependencies. At any time, three different views can be displayed and manipulated:

- The class charts;
- The clusters;
- The class descriptions.

For these three views a consistent internal structure is maintained. Any change made to one of the views is automatically propagated into the other two as soon as some "commit" operation is triggered.

The class chart view offers forms to fill out. The graphical view offers a drawing-tool interface, with a palette of graphical elements to be selected and placed on the screen. The layout of the graphical classes and clusters remains under the user's control.

The second EiffelCase component is aimed at helping the analyst to find and manage collections of classes. It is a large-scale browser that queries and updates a database of class information according to various criteria. The supporting repository is layered on top of the PCTE (Portable Common Tool Environment) Object Management System emerging as a CASE tools integration standard.

Acknowledgments

I am very grateful to Bertrand Meyer for his constant support and for suggesting bright and productive ideas. I also thank very much

Figure 8. Modified *VEHICLE* class description

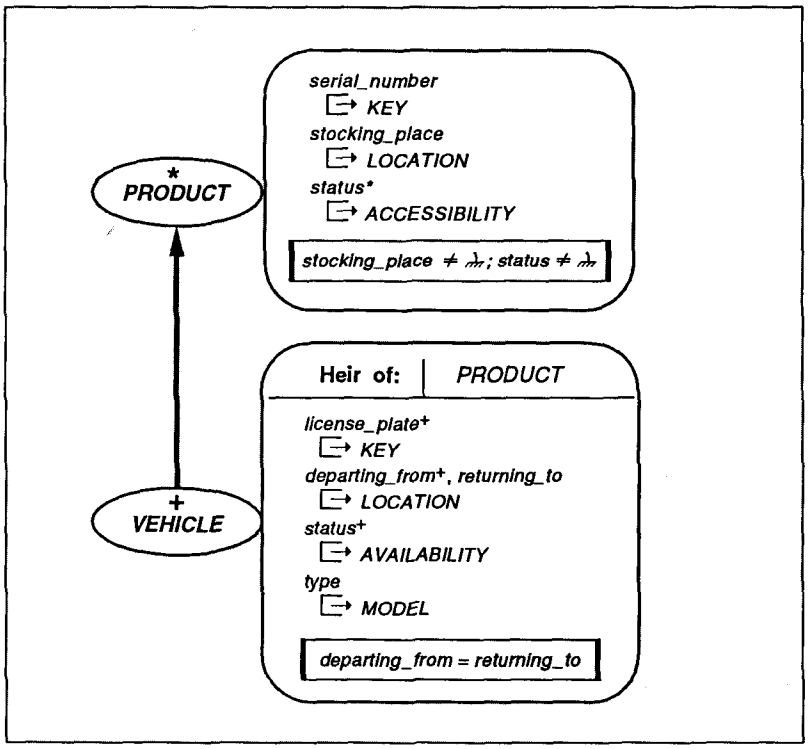


Table 7.
Summary of BON methodological steps

- DELINEATE THE SYSTEM BORDERLINE
- LIST CANDIDATE CLASSES OBSERVED IN THE PROBLEM DOMAIN
- GROUP CLASSES INTO CLUSTERS
- DEFINE CANDIDATE CLASSES IN TERMS OF QUESTIONS/COMMANDS/ CONSTRAINTS
- DEFINE BEHAVIORS: EVENTS, OBJECT COMMUNICATION PROTOCOLS, OBJECT CREATION CHART
- DEFINE CLASS FEATURES, INVARIANTS AND CONTRACTING CONDITIONS
- REFINE CLASS DESCRIPTIONS
- WORK ON GENERALIZATION
- COMPLETE AND REVIEW ARCHITECTURE

Kim Waldén and all the reviewers for their extremely helpful comments.

References

1. Beck, K., Cunningham, W. A laboratory for teaching object-oriented thinking. OOPSLA'89, Oct. 1989, pp. 1–6.
2. Booch, G. *Object Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc. 1991.
3. Brunet, J. Modeling the world with semantic objects. University of Paris I, Internal Report, 1991.
4. Cauvet, C., Roland, C., Proix, C. A design methodology for object oriented database. International Conference on Management of Data, Hyderabad, India, 1989.
5. Coad, P. and Yourdon, E. *Object Oriented Analysis,* Second Edition, Prentice-Hall, Englewood Cliffs, N.J., 1991.
6. Coad, P. Object-oriented patterns. *Commun. ACM 35,* 9 (Sept. 1992).
7. Duke, R., King, P., Rose, G., Smith, G. The object-Z specification language. In *Proceedings TOOLS 5,* (Santa Barbara, July–August 1991), Prentice Hall, Englewood Cliffs, N.J., 1991, pp. 465–483.
8. ESPRIT II, Business Class Technical Annex. Project #5311, Commission of the European Economic Community, Bruxelles, Sept. 1990.
9. Henderson-Sellers, B. *BOOK of Object-Oriented Knowledge.* Object Oriented Series, Prentice-Hall, Englewood Cliffs, N.J., 1991.
10. Meyer, B. Tools for the new culture: Lessons from the design of the Eiffel libraries. *Commun. ACM 33,* 9 (Sept. 1990), 69–88.
11. Meyer, B. *Eiffel: The Language.* Object Oriented Series, Prentice-Hall, Englewood Cliffs, N.J., 1992.
12. Meyer, B. Design by contract. In *Advances in Object-Oriented Software Engineering,* D. Mandrioli and B. Meyer, Ed., Object-Oriented Series, Prentice-Hall, Englewood Cliffs, N.J., 1991, pp. 1–50.
13. Monarchi, D., Puhr, G.I. A research typology for object-oriented analysis and design. *Commun. ACM 35,* 9 (Sept. 1992).
14. Nerson, J. Extending Eiffel toward O-O analysis and design. In *Proceedings TOOLS 5,* (Santa Barbara, July–August 1991), Prentice Hall, Englewood Cliffs, N.J., 1991, pp. 377–392.
15. Nerson, J. *Object-Oriented Architectures: Analysis and Design of Reliable Systems.* Prentice Hall, Englewood Cliffs, N.J. To appear.
16. Nierstraz, O., Tsichritzis, D., Gibbs, S. Component-Oriented software development. *Commun. ACM 35,* 9 (Sept. 1992).
17. Page-Jones, M., Constantine, L. and Weiss, S. Modeling object oriented systems: the uniform object notation. *Comput. Lang. 7,* (Oct. 1990), 69–87.
18. Potter, J. Software development with Eiffel, TOOLS 4 Tutorial Notes, Paris, June 1990.
19. Rubin, K., Goldberg, A. Object behavior analysis. *Commun. ACM 35,* 9 (Sept. 1992).
20. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. *Object Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, N.J., 1991.
21. Shlaer, S. and Mellor, S.J. *Object Oriented Systems Analysis, Modeling the World in Data.* Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, N.J., 1988.
22. Wasserman, A. I., Pircher, P.A., Muller, R.J. Concepts of object-oriented structured design. In *Proceedings TOOLS '89 Conference,* Paris, Nov. 1989, pp. 269–280.
23. Winblad, A.L., Edwards, S.D., King, D.R. *Object Oriented Software.* Addison-Wesley Publishing Company, 1990.
24. Wirfs-Brock, R., Wilkerson, B. and Weiner, L. *Designing Object-Oriented Software.* Prentice-Hall, Englewood Cliffs, N.J., 1990.

CR Categories and Subject Descriptors: D.2.1 [**Software**]: Software Engineering — *requirements / specifications;* D.2.10 [**Software**]: Software Engineering—*design;* I.6.0 [**Computing Methodologies**]: Simulation and Modeling—*general;* I.6.3 [**Computing Methodologies**]: Simulation and Modeling—*applications;* K.6.3 [**Computing Milieux**]: Management of Computing and Information Systems—*software management;* K.6.4 [**Computing Milieux**]: Management of Computing and Information Systems—*system management*

General Terms: Design, Experimentation

Additional Key Words and Phrases: Analysis, design, flexible software architecture, object-oriented notation and methodology, object-oriented software engineering, reliable component reusability, static class model and dynamic object model

About the Author:

JEAN-MARC NERSON is managing director of the Société des Outils du Logiciel (Paris). Current research interests include the analysis and design of reliable systems.

Author's Present Address: Société des Outils du Logiciel, 104 rue Castagnary, 75015 Paris, France; email: marc@eiffel.fr