(*Published as chapter 10 in "Handbook of Object Technology", CRC Press 1998*)

Business Object Notation (BON)

Kim Waldén, Enea Data, Sweden

Introduction

Business Object Notation, commonly known as BON, is a method for analysis and design of object-oriented systems, which emphasizes seamlessness, reversibility and software contracting. Its aim is to narrow the gap between analysis, design, and implementation by using the same semantic and conceptual base for the notation on all three levels. The idea is that if a method is to serve as support for the whole life cycle of a system, it must be possible to use its models and notations to do both forward engineering and reverse engineering. It must be possible not only to transform an initial problem model into an executable system, but also to reflect source code changes back into design and analysis.

Therefore, in BON you will not find the usual entity-relationship diagrams and state-charts that form the basis of nearly all analysis and design methods on the market today (whether claiming to be object-oriented or not). The reason is that no matter what you think about the expressiveness of these modeling concepts, they are incompatible with what is available at the implementation level, and so effectively prevent reversibility. Instead, BON relies on the power of the pure object-oriented modeling concepts: a system as a set of classes (abstractions) which are related by inheritance and client dependencies. The O-O concepts are combined with a clustering mechanism to group related abstractions, and with precise component specification based on strong typing and software contracts. The BON method is fully described in the book *Seamless Object-Oriented Software Architecture*, Prentice Hall 1995 [1]. It includes concepts and notations for static and dynamic modeling of object-oriented software as well as detailed guidelines for the modeling process.

The short overview in this chapter will concentrate on the general principles underlying the design of BON and the resulting concepts and notations, while there will only be room to touch upon the modeling process. Readers who want more detail are referred to the book above, in which more than 200 pages are devoted to the software process: describing the method guidelines and applying them in three extensive case studies drawn from practical usage.

Basic principles of BON

Generality

BON targets general software development, and does not attempt to cover every conceivable need that may arise for particular applications or with particular programming languages. Only concepts and notations that have proven useful in a wide spectrum of application domains are included in the method. Instead the user is free to complement the BON framework with additional descriptions and conventions that may be useful to a particular project. BON has been used successfully in many different application types, ranging from embedded technical systems to financial software and MIS applications.

With the advent of Java and CORBA techniques, large heterogeneous systems are becoming the norm rather then the exception. This requires a method that captures the essence of O-O abstraction without getting bogged down in details of specific language constructs. BON is independent of implementation platform, and has been used in projects targeting languages as different as Eiffel, C++, Smalltalk, and Object Pascal. In fact, the strong typing of BON has been appreciated in Smalltalk developments despite the fact that this language is not statically typed. This is because experienced Smalltalk programmers tend to think in terms of typed symbols, even if this is not reflected in the program code.

Seamlessness and reversibility

Using notations based on the same concepts and semantics throughout the development of a software product allows for a smooth transition from the initial analysis and design models down to executable code. However, what is even more important, it also allows the process to work in reverse: it should be possible to (automatically) extract abstract design views from existing implementations. Since the initial analysis and design is never correct in real projects, and many of the errors and shortcomings are not detected until implementation or testing time, the high-level models need to be continuously updated as the software evolves.

The big challenge is to keep the various models consistent with the code. This is usually not done in practice, since the cost is prohibitive with non-reversible methods. Therefore, as time passes, the design documentation will cease to be a correct description of the implementation and therefore will be of limited or no use to developers trying to understand the system. This in turn makes it much harder to maintain and enhance the software.

If we stick to classes as the main structuring mechanism and use only inheritance and client relationships to describe how the classes are related to each other, reversibility comes for free, because these concepts are directly supported by the major O-O programming languages. If, on the other hand, we include concepts that cannot be mapped to and from the eventual code, reversibility is lost. The conclusion drawn for BON is that reversibility is indeed necessary if a method is to play more than a marginal role in supporting software development [2].

The remaining question is then: is a notation based on class abstractions related through inheritance and client dependencies expressive enough? Can we do without the associations, multiplicities, and state-charts found in nearly all O-O analysis and design notations today? The answer is yes. Since BON is a different approach, you gain some and you lose some, but two crucial means of specification can more than compensate for the loss of free associations. These are strong typing and class contracts.

Typed interface descriptions

A class interface consists of the syntax and semantics of its applicable operations. The syntactic part of an operation, often called its *signature*, is the types of its return value and arguments, if any. Strong typing (also known as static typing) means that the class designer must specify the signature of all the services offered by the class. This permits automatic consistency checking already at the specification level, so mistakes can be detected early.

However, what is even more important is that typing is an essential conceptual aid in system modeling. Assigning types to names is really classification, which increases the information content in a specification considerably. Instead of relying on more or less vague naming conventions as to what kind of objects will be attached to a certain symbol at run-time, the reader of a typed specification can see this directly with no risk of misinterpretation. For example, if a class attribute refers to a list of trucks sorted on load capacity in a freight transport system, a typical attribute name in an untyped notation would be:

fleetOfSortedTrucks

which can be compared with the typed declaration:

fleet: SORTED_LIST [*TRUCK*]

The difference in precise information conveyed to the reader is considerable. With the second form we no longer have to guess what entities are described, but can find out their precise definitions by inspecting the classes *TRUCK* and *SORTED_LIST*.

Software contracting

Perhaps the most important principle in existence today for constructing correct, reliable software is the theory of Design by Contract [3]. The idea is to treat each subprogram as a subcontractor who undertakes to deliver some service (specified by a postcondition), but only if certain prerequisites are fulfilled (the precondition). Software contracts addresses two issues at once, both crucial for understanding large systems. First, the contract specifies formally what each component will do. Second, the responsibilities for taking care of unusual conditions are clearly separated between the supplier of a service and its clients.

As every experienced programmer know, it is the unusual cases, or even the ones that are not supposed to occur, that present the real problems in software engineering. Not knowing exactly who is responsible for the exceptional cases—the subprogram or its clients—fosters a desperate style of blind checking, also known as defensive programming. The seemingly innocent extra tests added "just in case" to improve system reliability not only fails to solve the problem, but in fact makes the situation worse. It leads to even less understandable software with more errors because of the complexity and confusion introduced by the added redundant code. This in turn calls for more extra tests, and so on.

With software contracts this does not happen, and each party can concentrate on its main task without having to worry about irrelevant detail. A contract protects both sides: it protects the client by specifying *how much* must be done and it protects the supplier by specifying *how little* is acceptable. This has much in common with standard practices in human societies, where a network of subcontractors is often employed to carry out a large undertaking.

Scalability

An important issue for any method and notation is the possibility to scale up from the small examples presented in textbooks to large, complex real-life systems. To this end we need two things: a facility to recursively group classes into higher-level units, and the possibility to zoom between the various levels of a large structure. The BON notation uses nested clustering and element compression to achieve this.

Classes may be grouped into clusters according to various criteria (part of a subsystem, heirs of a common ancestor, related functionality, etc.). Clusters and classes may in turn be grouped into new clusters on higher levels. Depending on the current focus of interest, many different views of the same underlying system need to be presented. Flat partitioning is not enough in this context, since we need to see different levels of detail in different parts of the system without losing track of how the parts fit into the big picture.

Therefore, most elements in BON may be *compressed*, which means that one or several graphical and textual elements are represented by a simpler element, its compressed form. Elements containing compressed forms may in turn be compressed. Since the level of compression can be independently selected for each structural element (recursively), the user may freely choose the amount of detail shown for each system part. A few simple changes in the compression levels may yield a dramatically different architectural diagram, making this approach well suited for automatic tools support.

Simplicity

In BON simplicity is a major guiding star. Contrary to many other approaches which try to include everything an analyst/designer might possibly need, BON tries to live by the maxim "small is beautiful". In our experience, if a modeling notation is to be of real use above what is already available in a structured development language such as Eiffel [4], it needs to be very simple and well-defined. It should be easy for a user to quickly master the whole notation, and its semantics and graphical symbols should be very clear and unambiguous.

For example, special care was taken to have very few different types of relation in BON (in fact only two static and one dynamic), and to craft the relationship symbols in a way that would rule out mistakes regarding the semantics and direction of each relation. Several modeling sessions with industrial users of other notations have show that this is a real problem. When consulting on object-oriented modeling, we usually begin by translating existing user diagrams into BON notation in order to better understand what they are aiming to do. When asked about details of complex diagrams, users very often become uncertain about the exact meaning of many of the relations, and whether A is related to B or if it was the other way around.

Space economy

The amount of information that can be conveyed to a reader of an architectural model of some part of a system—a cluster, a group of clusters, a group of related classes— depends strongly on how much can be fitted on one page (where a page is a terminal screen, a paper sheet or whatever can be observed in one glance). Breaking up an integral context into fragments that must be viewed one by one is detrimental to the overall readability.

Therefore, it is important for any notation designed to give global overviews of potentially large and complex structures to avoid wasting space. BON therefore provides compressed forms for all space-consuming graphical layouts. For example, it is too restrictive to have a full interface with operations and attributes as the only way to show a class. In BON the compressed form of a class is simply an ellipse enclosing its name and possibly some additional marker. Similarly, BON provides iconization of clusters and compression of relationships between classes into (fewer) relationships between clusters.

The static model

The static model describes the classes making up the system, their interfaces, how they are related to each other, and how they are grouped into clusters. It shows the system architecture and the contracts between each class component and its clients.

Informal charts

BON defines three types of informal static chart: the *System chart* listing the topmost clusters of the system, *Cluster charts* listing the classes and other clusters contained in each cluster, and *Class charts* describing the interface of each class. These are untyped charts meant to be used very early in the modeling process, particularly when communicating with non-technical people such as end-users, customers, or domain experts. An example of a class chart is shown in Figure 1. The chart contains a standardized header stating the type of chart (class in this case), its name, sequence number, a short description, and an indexing clause. The latter contains a number of keywords defined by the user, and one or more value entries for each keyword.

The class interface description is divided into three parts: *queries*, *commands*, and *constraints*. Queries are operations that return information about the system but do not change its state, while commands are operations that do not return information, but may change the system state. A clean separation of operations into queries and commands (which means that functions should not have side-effects) makes it much easier to reason about the correctness of classes. Finally, the constraints lists general consistency conditions and business rules to be obeyed.

Class charts are designed to facilitate later translation into typed interface descriptions, but also resemble formatted memos used to record various aspects of the problem domain. Therefore, it is not uncommon that class charts initially contain operations and constraints, which will later turn out to have no counterpart in a computerized system.

CLASS	*ELEVATOR*	**Part:** 1/1
TYPE OF OBJECT Models an elevator which is being pulled by a motor.		**INDEXING** **cluster:** *ELEVATOR_CONTROL* **created:** 1997-03-29 kw **revised:** 1997-04-08 kw
Queries	Current floor, Pending floor requests, Is elevator moving? Are doors open? Is elevator traveling up? Is elevator traveling down? Is elevator idle?	
Commands	Open doors. Close doors. Process floor requests.	
Constraints	Cannot be traveling both up and down. If stopped and no more requests then elevator is idle. Cannot move when doors are open. An idle elevator is not moving.	

Figure 1: Informal class chart

Class headers

The compressed form of a class is an ellipse containing the class name, possibly adorned by one or more of the graphical annotations shown in Figure 2. *Reused* classes are classes from earlier systems or third parties. *Persistent* classes are classes whose instances need to be stored externally. *Deferred* classes contain at least some operation that will only be implemented through descendant classes, while *effective* classes will be fully implemented. *Interfaced* classes contain external calls to the underlying software platform or to other systems. A *root* class is a class that gets instantiated when a system (or concurrent execution thread) is started.

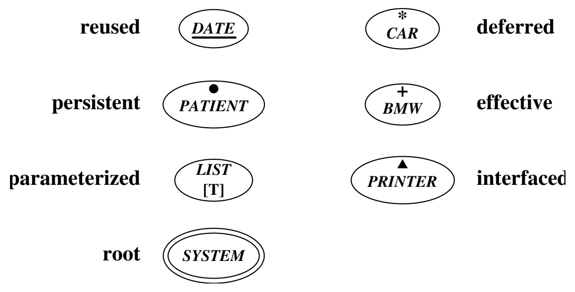


Figure 2: Class headers with annotations

Clusters

Classes can be recursively structured into clusters, whose graphical representation is rounded rectangles tagged with the cluster name, as shown in Figure 3.

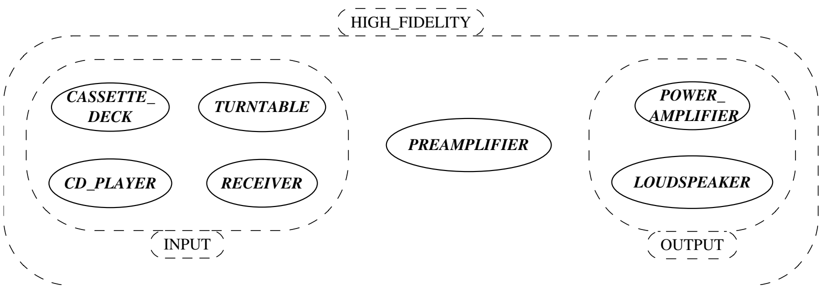


Figure 3: Nested clusters

Static relations

There are only two basic types of relation in BON, *client* and *inheritance*. Both are denoted by arrows: single arrows for inheritance and double arrows for client relations. The client relations have three varieties: plain client association, aggregation (part/whole), and shared association. Figure 4 show the different types of static relations.

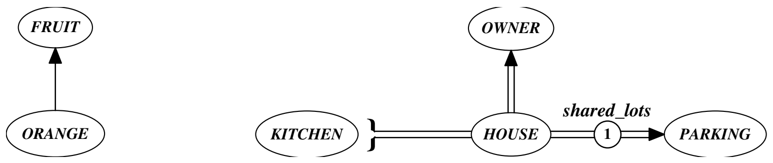


Figure 4: Inheritance and client relations

The double arrow ending in an open brace signifies that a kitchen is viewed as an integral part of a house, and the arrow marked by an encircled number states that all houses share one parking lot. The label above the circle is the name of the operation in class *HOUSE* giving rise to the relation.
A common modeling situation is that a class has an indirect client relation to another class through a generic (parameterized) container class, as shown in Figure 5. It is then seldom desirable to include the container class (usually a reused library class) at the same level as the classes related to the problem domain.

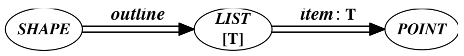


Figure 5: Generic supplier, expanded form

Therefore, BON provides a compact notation for such relations, as shown in Figure 6. Labels of type *LIST* […] are reminiscent of the semantic labels in entity-relationship diagrams, but instead of relying on natural language the precise semantics of the label can be found in the definition of the corresponding class.

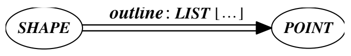


Figure 6: Generic supplier, compact form

Software contracts

The contract elements of object-oriented systems are *pre-* and *postconditions*, which apply to each operation of a class, and *class invariants*, which apply to the whole class. The precondition must be satisfied just before the operation is called (responsibility of the client), and the postcondition is then guaranteed to be satisfied when the call returns (responsibility of the supplier). The class invariant must be satisfied both before and after a call to any operation of the class, and represents general consistency rules defining the valid states for objects of this type.

Since operations may be inherited and redefined in object-oriented systems, a client can never be sure of what exact version will be invoked at runtime. For example, if a client needs to compute the areas of a list of geometric figures referred by an entity of type LIST [FIGURE], all that is available is the specification of the class FIGURE. The contract elements of this class must be enough to guarantee that whatever special figures may be in the list—now or in the future—the client call to compute its area will still be valid. Otherwise the contracting model breaks down.

Polymorphism with dynamic binding is the main key to software flexibility. It has the power to remove most of the discrete case analysis so error prone and vulnerable to future changes. However, flexibility is meaningless unless the resulting software is correct. Therefore, the following consistency rules are fundamental.

Laws of subcontracting:

- A descendant class must satisfy the class invariants of all its ancestors
- A descendant class may never weaken the postcondition of an inherited operation (since this would mean delivering less than specified by the ancestor)
- A descendant class may never strengthen the precondition of an inherited operation (since this would mean imposing restrictions on the client not specified by the ancestor)

Note that the above rules must be obeyed for every ancestor in case of multiple inheritance, and will therefore prevent combination of incompatible abstractions. This is crucial when building the complicated inheritance lattices needed by, for example, the general data structure libraries of typed language environments.

Typed class interfaces

Informal charts, such as in Figure 1, may be a good way to start analysis, particularly when non-technical people are involved, but the analyst/designer will soon need something more expressive. This leads us to the structured static diagrams with typing and software contracts. Three class interfaces taken from an elevator control system are shown in Figure 7. The interface of class *ELEVATOR* lists seven queries whose names and return types are separated by colon, and three commands with no return types. Lines starting with double dashes signify comments. The query *is_idle* specifies a postcondition, signaled by the framed exclamation point. The postcondition asserts that being idle is the same as being in neither of the states *is_traveling_up* and *is_traveling_down*. The commands *open_doors* and *close_doors* both specify as precondition (signaled by the framed question marks) that the elevator must not be moving when these operations are called.

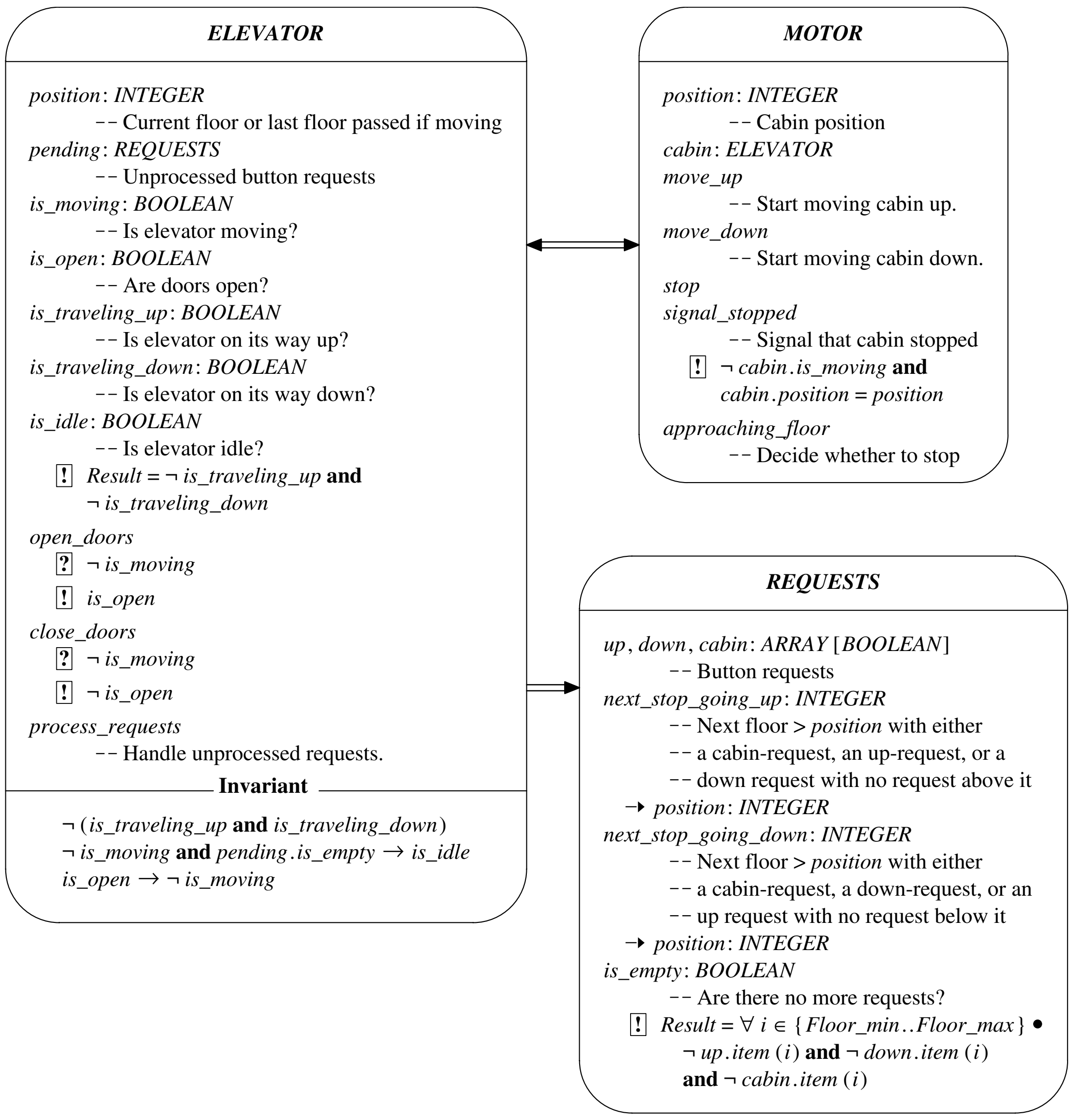

ELEVATOR
position: INTEGER
-- Current floor or last floor passed if moving
pending: REQUESTS
-- Unprocessed button requests
is_moving: BOOLEAN
-- Is elevator moving?
is_open: BOOLEAN
-- Are doors open?
is_traveling_up: BOOLEAN
-- Is elevator on its way up?
is_traveling_down: BOOLEAN
-- Is elevator on its way down?
is_idle: BOOLEAN
-- Is elevator idle?
! Result = ¬ is_traveling_up and ¬ is_traveling_down
open_doors
? ¬ is_moving
! is_open
close_doors
? ¬ is_moving
! ¬ is_open
process_requests
-- Handle unprocessed requests.
Invariant
¬ (is_traveling_up and is_traveling_down)
¬ is_moving and pending.is_empty → is_idle
is_open → ¬ is_moving
MOTOR
position: INTEGER
-- Cabin position
cabin: ELEVATOR
move_up
-- Start moving cabin up.
move_down
-- Start moving cabin down.
stop
signal_stopped
-- Signal that cabin stopped
! ¬ cabin.is_moving and cabin.position = position
approaching_floor
-- Decide whether to stop
REQUESTS
up, down, cabin: ARRAY [BOOLEAN]
-- Button requests
next_stop_going_up: INTEGER
-- Next floor > position with either
-- a cabin-request, an up-request, or a
-- down request with no request above it
→ position: INTEGER
next_stop_going_down: INTEGER
-- Next floor > position with either
-- a cabin-request, a down-request, or an
-- up request with no request below it
→ position: INTEGER
is_empty: BOOLEAN
-- Are there no more requests?
! Result = ∀ i ∈ {Floor_min..Floor_max} • ¬ up.item (i) and ¬ down.item (i) and ¬ cabin.item (i)

Figure 7: Interface of ELEVATOR classes

The class invariant specifies a number of consistency conditions for all elevator objects. An elevator cannot be traveling both up and down at the same time. An elevator that does not move and have no pending requests is in idle state. An elevator cannot move if the doors are open.

In class *REQUESTS* the queries *next_stop_going_up* and *next_stop_going_down* each takes an argument signaled by the small horizontal arrow. The query *is_empty* uses the predicate logic notation of BON to specify its semantics. According to the postcondition, being empty means that for each floor number taken from the interval between the lowest and highest floor serviced by the elevators, there is no button request in neither of the queues for up requests, down requests, or cabin requests.

Static architecture

A possible static architecture for the elevator control system is shown in Figure 8. Comparing this diagram to the interface specifications in Figure 7, shows that we get many of the client relations for free, since they can be derived directly from the return types of the queries defined in the classes *ELEVATOR* and *MOTOR*.

.

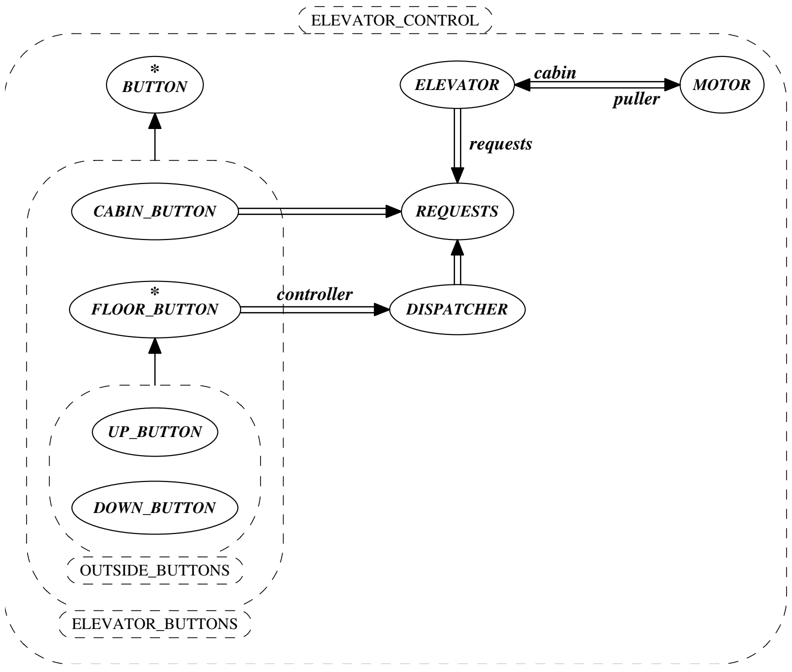


Figure 8: Elevator system architecture

The dynamic model

Informal charts

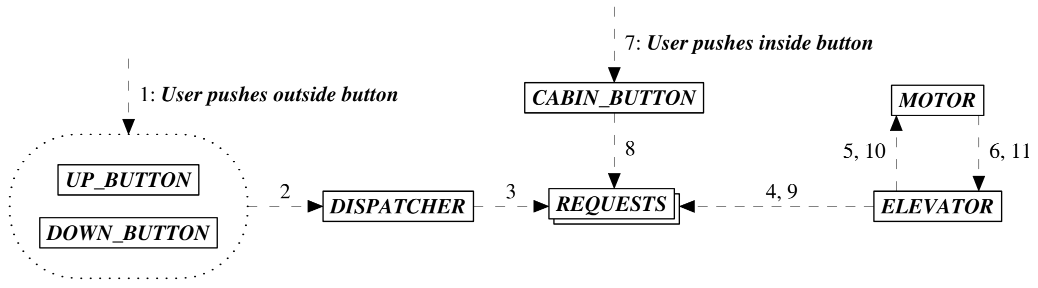
There are three types of informal chart for dynamic modeling in BON: *Event charts* listing incoming external events and outgoing internal events, *Scenario charts* listing interesting system usage, and *Creation charts* listing the classes responsible for creating instances of problem domain classes in the system. Figure 9 shows a scenario chart with three potentially interesting scenarios selected for further investigation. Each scenario has a short phrase for quick reference, and a description explaining the scenario a little more.

SCENARIOS	*ELEVATOR CONTROL*	**Part:** 1/1
COMMENT Set of representative scenarios to show important types of system behavior.	**INDEXING** **created:** 1997-04-13 kw	
Visit friend in neighboring apartment: A person leaves her apartment, calls an elevator and goes to see a friend living at another floor level.		
Coming in and going out in parallel: One person enters ground floor, calls elevator, and goes up, while another person enters a floor, calls elevator, and goes down.		
Partly joint ride: One person enters elevator at 5th floor and goes to 2nd, while another person enters at 3rd and goes to basement.		

Figure 9: Scenario chart

Dynamic diagrams

The first of these scenarios is illustrated in the dynamic diagram showed in Figure 10.



	Scenario: Visit friend in neighboring apartment
1	*User enters floor at own apartment level and pushes floor button.*
2	*Button asks dispatcher to accept request.*
3	*Dispatcher chooses suitable elevator and enters a request in that elevator's queue.*
4, 5	*Chosen elevator takes request from queue and tells motor to move in corresponding direction.*
6	*Motor signals that cabin has stopped at requested floor.*
7	*User enters cabin and pushes button to floor where friend lives.*
8	*Request is entered in elevator's queue.*
9, 10	*Elevator takes request from queue and tells motor to move.*
11	*Motor signals stop and user leaves elevator to visit friend.*

Figure 10: Dynamic object scenario

It consists of a number of objects calling each other (sending messages in Smalltalk terminology). Each object is labeled by its type (name of corresponding class), and a parenthesized identifying label if needed. A rectangle signifies a *single object*, while a double rectangle refers to a *set of objects* of the given type.

A dashed arrow represents a message passed from one object to another, and is called a *message relation*. The arrows are labeled by sequence numbers showing the order in which the messages are sent, and a *scenario box* adds a general free text description of the corresponding actions. Objects may be collected into object groups (dotted rounded rectangles), and messages passed to or from such a group then represents messages to or from one or more of the objects in the group.

A message relation is always potential, since conditional control (such as multiple choice or iteration) is not included in dynamic diagrams, and all data flow resulting from values being returned is also implicit. For reasons of simplicity and abstraction, the diagrams show only passing of control. In case we need to express more, a suitable dynamic modeling notation can be used in the text entries of the scenario boxes.

The method

The BON approach contains a set of guidelines regarding what to build—the BON *deliverables*: informal charts, static architecture, class interfaces, dynamic scenarios—and how and when to do it. Nine standard tasks are defined to be carried out in an approximate order, which represents an ideal process [5]. Nine standard activities are also defined, some or all of which may be involved in each of the tasks.

The BON process is not meant to be followed to the letter. Rather its purpose is to guide: guide managers by providing a bench mark against which progress can be measured, and guide the designer when at loss for what to do next. Here we have only room to briefly list the tasks and activities, while a full discussion along with examples of practical usage can be found in [1].

BON process tasks

The tasks are grouped into three parts: gathering information about the problem, elaborating what has been gathered, and designing a computational model.The first part involves separating what should be modeled from what can be left out; extracting an initial set of candidate classes from the concepts used in the problem domain; selecting classes from this set, group them into clusters and sketch principal similarities (inheritance) and collaborations through client dependencies.
The second part involves defining classes in terms of queries, commands, and constraints; sketching system behavior in terms of relevant object scenarios; defining public features using typed interfaces and formal contracts. Finally, the third part involves refining the problem domain architecture with new design classes and new features; factoring out common behavior into deferred classes; completing and reviewing the full system architecture.

GATHERING ANALYSIS INFORMATION

1. Delineate system borderline
2. List candidate classes
3. Select classes and group into clusters

DESCRIBING THE GATHERED STRUCTURE

4. Define classes
5. Sketch system behavior
6. Define public features

DESIGNING A COMPUTATIONAL MODEL

7. Refine system
8. Generalize
9. Complete and review system

BON standard activities

Of the nine standard activities, the first four are continuously repeated during both analysis and design. The fifth occurs mostly during analysis, but may also be repeated during design of large systems. The last four are chiefly part of the design task.

ANALYSIS AND DESIGN

1. Finding classes
2. Classifying
3. Clustering
4. Defining class features
5. Selecting and describing object scenarios

DESIGN

6. Working out contracting conditions
7. Assessing reuse
8. Indexing and documenting
9. Evolving the system architecture

Conclusion

It is our experience that most developers who use analysis and design methods systematically (a small minority to begin with), use them only as help to produce an initial system design. Whether or not the resulting models turn out to be useful, they usually lose their importance after the first software release (if not before), and the implementation code takes on a life of its own. The BON method has proved that there is a way out of this. By providing a conceptual framework for analysis and design based on pure O-O abstraction with software contracts, the necessary reversibility can be achieved at low cost.

Without reversibility it is impossible, in the long run, to ensure consistency between specification and implementation. As long as modeling concepts such as free associations and state diagrams, which are incompatible with implementation languages, continue to be included as essential ingredients in analysis and design methods, reversibility is lost. The goal of BON has been to break this tradition and take the potential of O-O modeling to its logical conclusion.

References

1. Kim Waldén and Jean-Marc Nerson, *Seamless Object-Oriented Software Architecture*, Prentice Hall, Englewood Cliffs, N. J., 1995, pp. 438.

2. Kim Waldén, “Reversibility in software engineering,” *IEEE Computer*, vol 29, no 9, Sep. 1996, pp. 93–95.

3. Bertrand Meyer, *Object-Oriented Software Construction*, 2nd edn, Prentice Hall, Englewood Cliffs, 1997, pp. 331–438.

4. Bertrand Meyer, *Eiffel: The Language*, Prentice Hall, Englewood Cliffs, 1992

5. David Lorge Parnas and Paul C. Clements, “A rational design process: how and why to fake it,” *IEEE Trans. Software Engineering* **SE-12**(2), Feb. 1986, pp. 251–257