An Introduction to BON

Richard Paige

Department of Computer Science, York University

`paige@cs.yorku.ca`

July 12, 1998; Revised August 6, 1999

Introduction

As designers and programmers of software, we need *notations* that we can use to write down our ideas. When we write programs, we use a programming language--like Eiffel--to describe how we want the machine to behave. A programming language is a useful notation for writing down *algorithms*, and for efficiently implementing systems. However, when we are trying to solve a problem (one that will eventually be implemented as a computer program) a programming language isn't likely to be the most appropriate notation. When we are problem solving, we probably won't be concerned with the technical details that we must worry about when we are programming. When we are problem solving, and when we are designing our systems, we want to be able to write down *abstractions*, using a notation that is well-suited to expressing abstractions. When we are thinking, we don't want to be forced to think in terms of programming language idioms--things like registers, `while` loops, pointers, and the like. We usually want to think in terms of abstractions--high-level concepts like classes, relationships, components, and so on. And there are better notations for describing these abstractions than programming languages.

If a programming language is appropriate for implementation, but not always for analysis and design, then we likely need a different notation for these latter tasks. We now briefly summarize one such notation, which will be used in lectures and assignments.

The Business Object Notation

The *Business Object Notation*, or **BON** for short, is a graphical notation for describing object-oriented programs and systems. BON is useful, because it is very simple, because it allows very compact descriptions of systems, and because it supports most of the styles of description that are necessary for talking about object-oriented systems. BON works seamlessly with the programming language Eiffel, but can work easily with other programming languages, like Java or C++.

Throughout this short document, we present examples of the main BON notations. This guide is not meant to be a complete description of the facets of BON; you are referred to the references for more details.

A question that you may be asking at this point is: why bother with BON, when there are so-called ``industrial-strength'' notations like UML and OMT available? One problem with these notations is that they are very large and very complex--in fact, they are too complex to cover in any real detail in a one-term course that must also cover many other topics. A significant reason for using BON in this course is that it is very simple and very concise; it is feasible to discuss all of BON within the scope of a couple of lectures. BON is a good notation for learning the concepts behind object-oriented technologies.Once you have an understanding of the key object-oriented concepts, like classes, objects, and inheritance, then you may decide to move to different notations. But BON also has significant technical advantages over other modeling languages, like UML; reference [2] has some details on these.

Terminology

Before continuing, we briefly review some relevant terminology.

A *class* is the fundamental construct in object-oriented programming. A class is, informally, a type and a module that encapsulates **features**. A feature may be an *attribute* or a *routine*. An attribute is associated with state; think of it as a variable local to an occurrence of the class. A routine is either a function or a procedure. A function calculates and returns a value, while a procedure changes state and doesn't return a value.

An *object* is a variable with a class for its type. An object is usually said to be an *instance* of a class. Each object has instances of the attributes that belong to its class type, and has instances of the routines that belong to the class type[1].

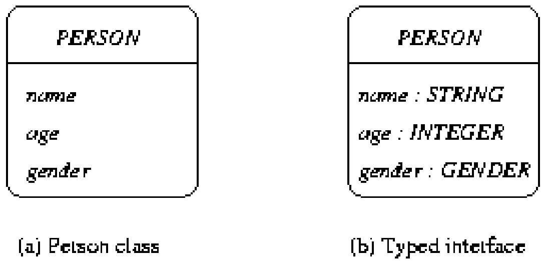
A *client* of a class *C* is another class that *uses C*. Interesting programs are usually made up of a number of classes that interact via *client relationships* (more on this shortly).

Describing Classes and Objects

The fundamental notation in BON is that for describing classes. To explain the BON notation for these concepts, we reformulate a standard example, a class *PERSON*.

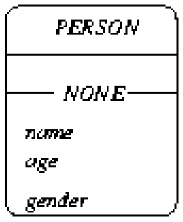
The class *PERSON* has three features. Fig. 1(a) shows the BON specification of the class. The name of the class, in all capitals, is given at the top of the diagram[2]. The three features are *name*, *age*, and *gender*. By default, these features are all public and visible, and can therefore be accessed and used by any clients of *PERSON*. We haven't yet specified any *types* for these attributes. If we wanted to provide types, we would write the types *after* the features name, with a colon in between the name and the type. An example is in Fig.1(b).

Figure 1: Variants of a person class



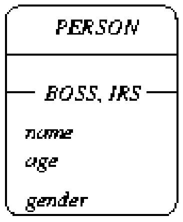
We may want to specify that the three features are not visible to all clients of the class. To do this in BON, we draw the class *PERSON* again, but this time we also write in the class the list of *clients* (other classes) that can access the features. Since we do not want to allow any clients to access the features, we write *NONE* for the client list, indicating that the features are *private.*

Figure 2: Class with private, but no public features



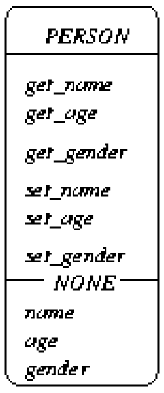
If we want to specify that selected classes, e.g., *BOSS* and *IRS*, can access the features of *PERSON*, then we write the names of these classes on the access list. This is shown in Fig.3.

Figure 3: Class with an access list



We may want to specify a class with some private features - inaccessible to clients directly - but with some public routines that can be used by clients to change or reveal the private features. An example of such a situation is in Fig.4, where we have refined our description of class *PERSON* to have private attributes and several routines that can change or reveal the values of attributes.

Figure 4: Class with public routines, private attributes



Clients of the class *PERSON* can access the routines *get_name*, *set_name*, et cetera, but cannot directly access the attributes *name, age, gender*, because they are not on the access list in Fig. 4. By default, the routines above the private section of the class are accessible by all clients. This flexibility--i.e., being able to specify particular classes that can access attributes--is more powerful than what is available in languages like Java and C++ (though C++ can express selective access by `friend` classes).

Sometimes, especially when we are drawing BON diagrams with lots of classes, we don't want to describe the features of a class in the diagram; the name of the class, and its relationships with other classes, are more important to describe. To this end, BON has a *compressed form* for a class - an ellipse containing just the name of a class. We will see examples soon, but the compressed form is ideal for use when we don't want to focus on the details of a class.

We may want to specify more than classes: we may want to specify *objects*, which are variables of a particular class. In Eiffel, we specify objects by declaring variables of a specific class. For example, if *PERSON* was an Eiffel class, we could declare a variable *fred*, of class *PERSON*, by writing

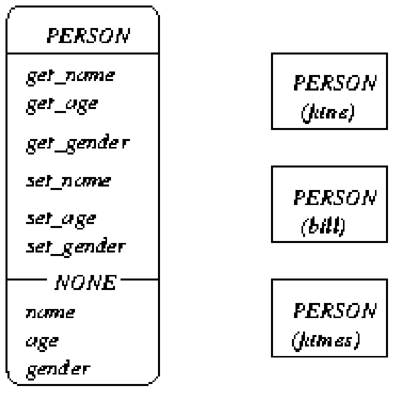
```
fred : PERSON
```

In BON, we write objects (which are variables of a class) as rectangles, with the *name* of the class at the top of the rectangle, and the name of the variable in brackets below the class name. If there is only one object of the class that we are interested in, we can omit the name of the variable. Fig. 5 shows an example. Objects *jane, bill*, and *james* are all variables of class *PERSON*. In Eiffel, we would express this diagram as

```
jane, bill, james : PERSON
```

to declare that `jane, bill` and `james` are all objects of class `PERSON`.

Figure 5: Objects associated with class *PERSON*



Contracts

Software contracts -- an important facility for developing reliable and robust software components -- are *preconditions* and *postconditions* associated with routines. Recall that a precondition of a routine is a boolean expression that states what must be true of the inputs to a routine. The postcondition is a boolean expression that states what must be true after the routine has executed. Here is an example of a contract, written in Eiffel. The example consists of a contract and an implementation for a procedure `remove` that belongs to a class that contains a `list`.

```
remove is
  require  not list.is_empty
  do
    list := list.tail
  ensure  list = old (list.tail)
  end -- remove
```

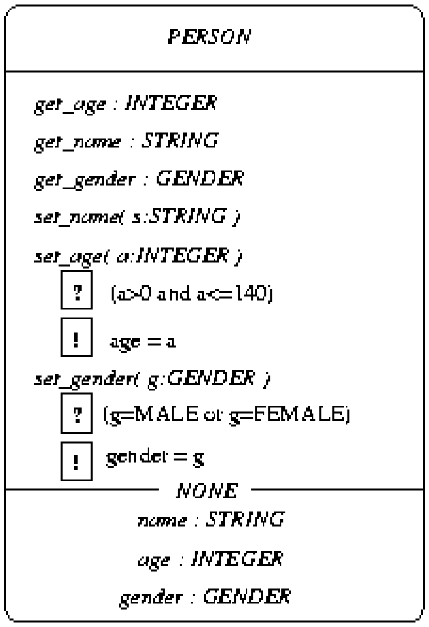
The precondition (the **require** clause) states that the list cannot be empty; the postcondition (the **ensure** clause) states that when the procedure is finished, the value of `list` must be the original value of the `list.tail`.

The **old** keyword can be used in Eiffel postconditions. Think of **old** as a function that can be applied to *any expression*. The value of **old** *expr* is the value of *expr* when the routine was called. **old** is very useful in expressing *changes* in variables. For example, the effect of an assignment `x:=x+1` can be specified as

```
ensure x = old(x) + 1
```

BON supports writing contracts: you can supply preconditions and postconditions with your routines in the descriptions of the class. Fig. 6 contains an example. Class *PERSON* is refined so as to add contracts to routines **setAge** and **setGender**. The precondition for a routine is labelled with a question mark, ?, in a box, while the postcondition is labelled with an exclamation mark, !, in a box. (Alternative textual labels sometimes use the words `pre` and `post`, or `require` and `ensure`, instead of the boxed punctuation marks.)

Figure 6: Class *PERSON* annotated with contracts and types



In the BON description, we have supplied contracts for some routines, but not others. We have added interfaces for the routines, and type information for each of the attributes. By providing type information, we have more ways to verify the correctness of our contracts.

The BON assertion language, used to write contracts, is based on typed predicate logic. It uses the standard predicate logic operators and quantifiers: $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**), $\forall$ (**for_all**), etc. In general, the text forms of operators are used in textual BON, and the graphical forms in the diagrams.

The basic form of a quantified expression in BON is:

$$\mathit{Quantifier\ Range} \ \mid \mathit{Restr} \bullet \mathit{Prop}$$

Quantifier is one of $\forall$ and $\exists$. Prop is a proposition, e.g., $x > 0 \wedge y$. The range can be a type range (i.e., each variable in a list is of a given type) or a member range. Here are examples.

- *Type range:* $\forall v : \mathit{VEHICLE};\ p : \mathit{PERSON}$
- *Member range:* $\forall c \in \mathit{children}$ (where *children* is a set).

Restr is a restriction on the type range, e.g.,

$$\forall i : \mathit{INTEGER} \mid i > 5 \ldots$$

can be read as ``for all integers *i* such that *i* is greater than 5".

In propositions, the colon operator : can be used to ask if a variable is of a given type, e.g.,

$$\forall p : \mathit{PERSON} \bullet p : \mathit{UNDERGRAD}$$

which asks if *p* is of type *UNDERGRAD* (where *UNDERGRAD* is a descendent of class *PERSON*).

Here is an example of a complete assertion.

$$\forall x : \mathit{INTEGER};\ p : \mathit{PERSON} \mid x > 2 \bullet p : \mathit{TODDLER} \wedge p.\mathit{age} > x$$

The quantifier introduces bound variables *x* and *p*, with the added restriction that integer *x* is greater than 2. The scope of the quantifier is a proposition which states that *p* is of type *TODDLER* and query *p.age* is greater than *x*.

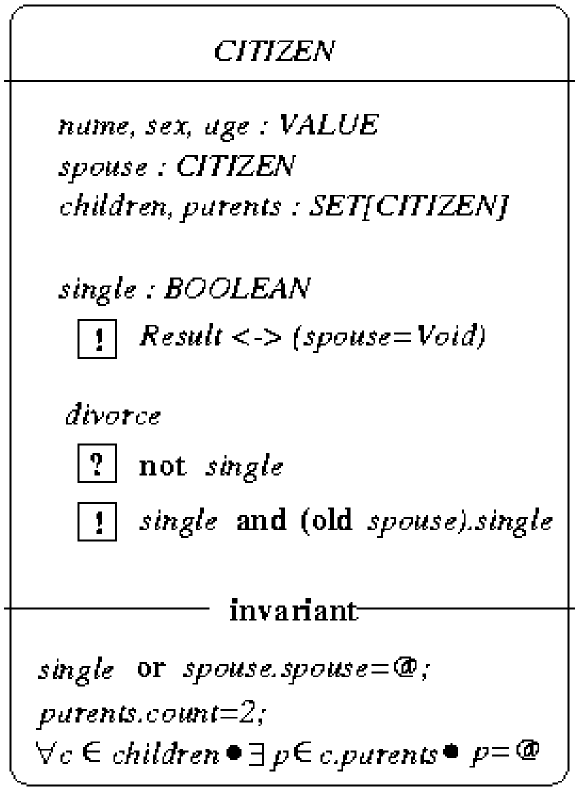
BON provides a further kind of assertion that we haven't yet seen: the *class invariant*, which describes properties - using boolean expressions - that apply to the class as a whole. We will see an example of a class invariant in the next subsection.

Graphical and Textual Forms

So far, we have seen the use of BON for writing graphical specifications. BON also has a textual notation, which is entirely equivalent to the graphical notation. That is, if a developer writes down a graphical BON specification, there is an equivalent textual BON specification. The benefit of having such an equivalence is in making the move to implementations. Programs are usually written in a textual form, so if we use BON's graphical notation, we need to be able to transform the BON graphical specifications into a text form as easily as possible.

We demonstrate the equivalence with an example, in the process providing a further example of using contracts. The example also demonstrates a further BON feature-the *class invariant*. Fig. 7 shows a graphical specification.

Figure 7: BON graphical specification


CITIZEN
name, sex, age : VALUE
spouse : CITIZEN
children, parents : SET[CITIZEN]
single : BOOLEAN
! Result <-> (spouse=Void)
divorce
? not single
! single and (old spouse).single
invariant
single or spouse.spouse=@;
parents.count=2;
∀c ∈ children ● ∃ p∈ c.parents ● p=@

The equivalence textual specification is as follows.

```
class CITIZEN feature
   name, sex, age : VALUE
   spouse : CITIZEN
   children, parents : SET[CITIZEN]

   single : BOOLEAN  ensure Result<->(spouse=Void) end
   divorce
     require  not single
     ensure   single and (old spouse).single
   end

   invariant
     single or spouse.spouse=Current;
     parents.count=2;
     for_all c member_of children it_holds
       (exists p member_of c.parents it_holds p=Current)
end -- class CITIZEN
```

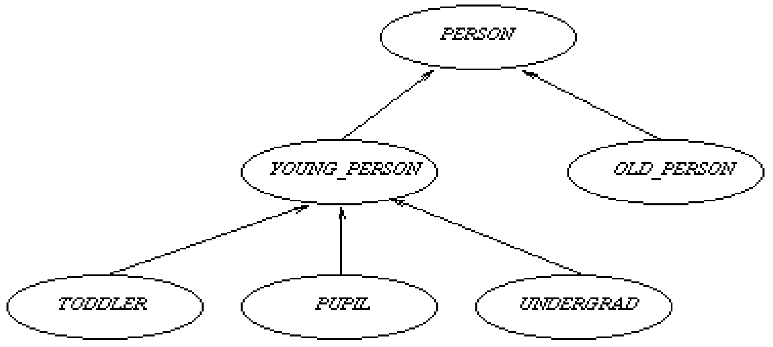
The class invariant (denoted using the **invariant** clause) gives a predicate that must be maintained by all objects of the class. Every routine of the class[3] can assume, when it is called, that the invariant is true, and every routine must establish, when it terminates, the class invariant[4].

Inheritance

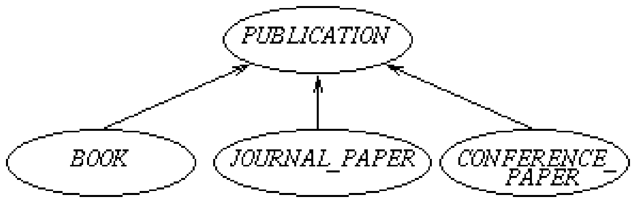
Inheritance is a mechanism for reuse of classes. It is a way to add new features to a class, or to extend a class, without having to change any of the clients that use the class. Inheritance is the way that designers can produce classes that satisfy a very important software engineering principle, the *open-closed* principle. Designers want classes that are *open*, in the sense that they are reusable and can be extended to have new features. But designers also want to have classes that are *closed*, in the sense that clients can use them and rely on them not changing.

The BON diagrams for inheritance are very similar to those used in other object-oriented and visual notations. In inheritance diagrams, classes are often written in a compressed form, omitting routines and attributes (especially if the class is very large). Inheritance is drawn as an arrow, directed from the child class (that which is inheriting) to the parent class. Figures 8(a) and 8(b) show examples.

Figure 8: Inheritance diagrams in BON



(a) A person hierarchy

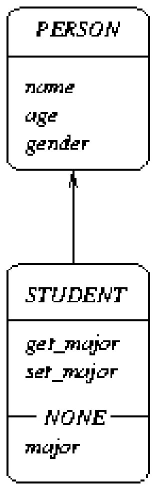


(b) A publication hierarchy

Remember that inheritance can be used to describe the **is-a** relationship: in an inheritance hierarchy, the child class **is-a** parent class. So, in Fig. 8(a), *TODDLER*s are (is-a) *YOUNG_PERSON*s, and *PUPIL*s are (is-a) *YOUNG_PERSON*s, et cetera. When setting up an inheritance relationship between two classes, always ask yourself: is the child class really an example of the base class too?

Inheritance relationships can also be drawn between the detailed visual presentations of classes that we saw earlier. For example, Fig. 9 shows an inheritance relationship between *PERSON* and *STUDENT*: a *STUDENT* **is-a** *PERSON*. We have drawn the arrow between the two classes, rather than just the elliptical short forms. In the process, we show some of the routines and attributes that the classes offer.

Figure 9: Detailed view, with inheritance



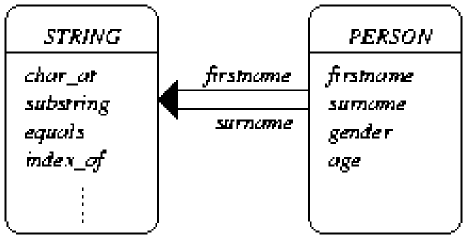
For large systems, we will use the short form when describing inheritance relationships, because otherwise the diagrams will get cluttered and hard to understand. For drawing inheritance relationships among only a few classes, it is usually not difficult to understand the diagram if the long form notation is used.

Client-Supplier Relationships

BON offers one further kind of relationship that can be drawn between classes: the *client-supplier* relationship. The notion of a client-supplier relationship is very simple: there are two classes, one the client, and the other the supplier. The client *uses* some facility of the *supplier* class. More often than not, the client may have an attribute that is of the supplier type, but the relationship can also be used to indicate the use of a routine of the supplier.

BON actually possesses three kinds of client-supplier relationships. The most important one is the *association* relationship (we talk a bit about the other two shortly). An example of an association is shown in Fig. 10. The class *PERSON* is a client of the class *STRING*. The double-line arrow from *PERSON* to *STRING* indicates the client-supplier relationship. The class at the tail of the arrow is the client, and the class at the head of the arrow is the supplier. In this case, the relationship is such that *PERSON* is a client of supplier *STRING*. The relationship is defined so that *PERSON* has two attributes, *firstname* and *lastname*, both of class *STRING*. The attributes of the *PERSON* class that are of class *STRING* are written on the client-supplier arrow.

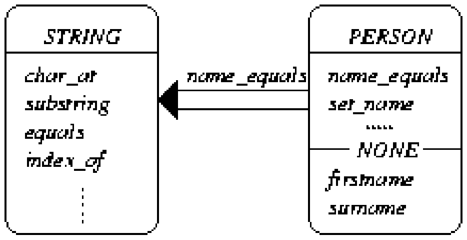
Figure 10: Association relationship between two classes



In this particular example, the association arrow is indicating a **has-a** relationship between *PERSON* and *STRING*: class *PERSON* **has-a** *STRING*.

A different example of an association relationship would be where class *PERSON* uses routine *equals* of class *STRING*. This is shown in Fig. 11.

Figure 11: Association, use of routine

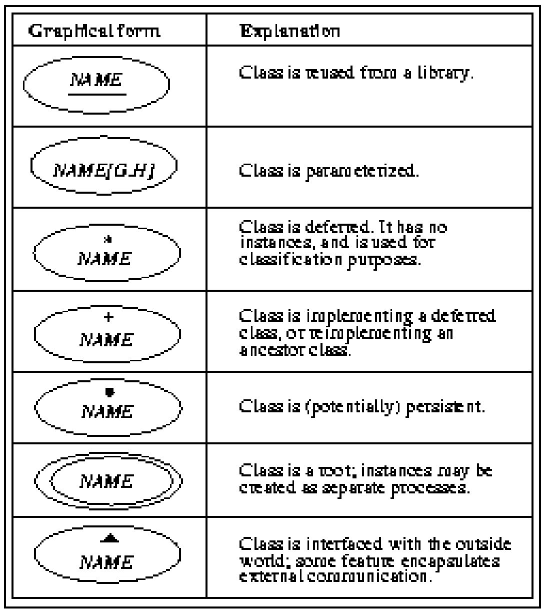


The routine *nameEquals* of class *PERSON* seemingly makes use of some feature of class *STRING*.

Class Header Annotations

BON class diagrams can be further extended with header annotations, designed to give more information about the class at the abstract level. There are several forms of class header annotation, but the most important ones are shown in Fig. 12.

Figure 12: Class header annotations



Graphical form	Explanation
NAME	Class is reused from a library.
NAME[G,H]	Class is parameterized.
* NAME	Class is deferred. It has no instances, and is used for classification purposes.
+ NAME	Class is implementing a deferred class, or reimplementing an ancestor class.
● NAME	Class is (potentially) persistent.
NAME	Class is a root; instances may be created as separate processes.
▲ NAME	Class is interfaced with the outside world; some feature encapsulates external communication.

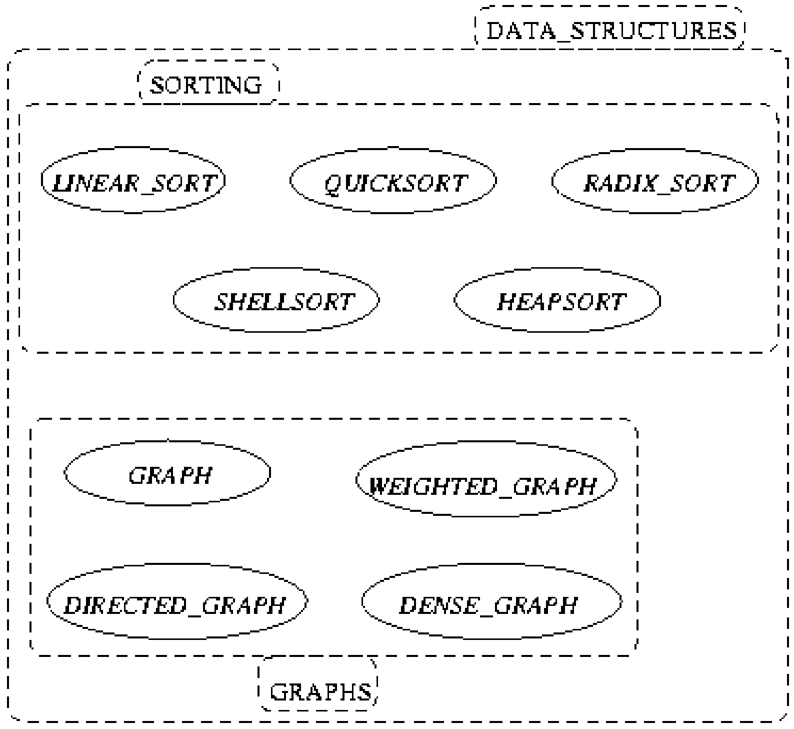
With such annotations, a reader of your class diagrams can learn more about the nature of the class, without having to read the details of the class itself.

Clusters

A *cluster* represents a group of related classes (and possibly other clusters) according to some point of view. Classes can be clustered depending on the particular characteristics of a system a designer wants to highlight.

Fig. 13 shows an example of a cluster. Clusters of classes (possibly with relationships among the classes) are encompassed in a dashed rounded rectangle. The rectangle is tagged with the name of the cluster. This name can then be used to refer to the cluster in other diagrams.

Figure 13: A nested cluster



Advanced Notions

So far, we have seen the basic features of BON. This section discusses some of the more advanced concepts of BON.

Inheritance involving clusters

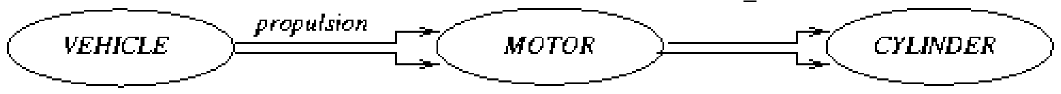
The inheritance relation between classes can be generalized to apply to clusters. There are three cases. In the following, let an element be a cluster or a class.

1. If all elements in a cluster *X* inherit from an element *A* outside of *X*, then *X* inherits from *A*. One link can be drawn from *X* to *A*.
2. If an element *A* outside cluster *Y* inherits from all elements in *Y*, then *A*inherits from *Y*. All links from *A* to elements in *Y* can be compressed into one link from *A* to *Y*.
3. An inheritance link between two elements can be compressed into being hidden.

Aggregation

Earlier, we saw association relationships between classes. As we said, these are only one kind of client-supplier relationship in BON. To be precise, there are three types of client-supplier relationships: *association, shared association,* and *aggregation.* We have seen the association relationship so far. Aggregation is new. An aggregation relation between a client class and a supplier class means that each client instance may be attached to one or more supplier instances which represent integral parts of the client. Aggregation is an important semantic concept, so it has a special notation in BON. Aggregation links are drawn as double lines (like client-supplier) but instead of ending in a single arrow, end in a *double arrow.* Fig. 14 shows an example.

Figure 14: Aggregation relations



In the diagram, feature *propulsion* of class *VEHICLE* is attached to one or more instances of class *MOTOR*. Feature *combustion_chamber* of class *MOTOR* is attached to one or more instances of class *CYLINDER*. Informally, we can say that *MOTOR* is a `part-of' a *VEHICLE.*

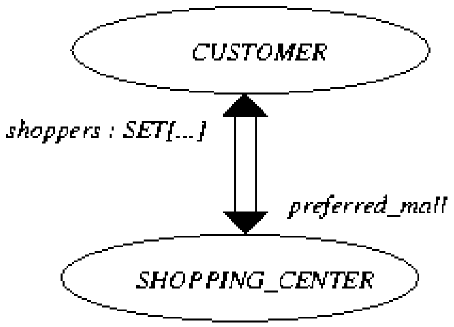
So what are the differences between association and aggregation? There are two important differences. The first is that with aggregation, deletion of the client implies deletion of the supplier. For example, suppose we had an instance of *MOTOR*, and we decided to get rid of it (in an Eiffel program, we would say that the instance is `freed' or `garbage collected'). The *MOTOR* object has, as a part, instances of *CYLINDER* objects. These would be deleted as well.

The more important difference is more technical. When mapping BON to Eiffel, aggregation relationships can be mapped to use of *expanded types*. Effectively, that means that the supplier is not connected to the client via a *reference*.

Bidirectional links

A set of client-supplier associations in each direction between two classes may be combined into a double link with an arrowhead at each end. Fig. 15 has an example.

Figure 15: Bidirectional association links



A convention is needed to show which relation each label refers to. The rule is to put the label closer to the supplier side. In Fig. 15, *preferred_mall* is a feature in *CUSTOMER* of type *SHOPPING_CENTER* while *shoppers* is a feature in *SHOPPING_CENTER* of type *CUSTOMER*.

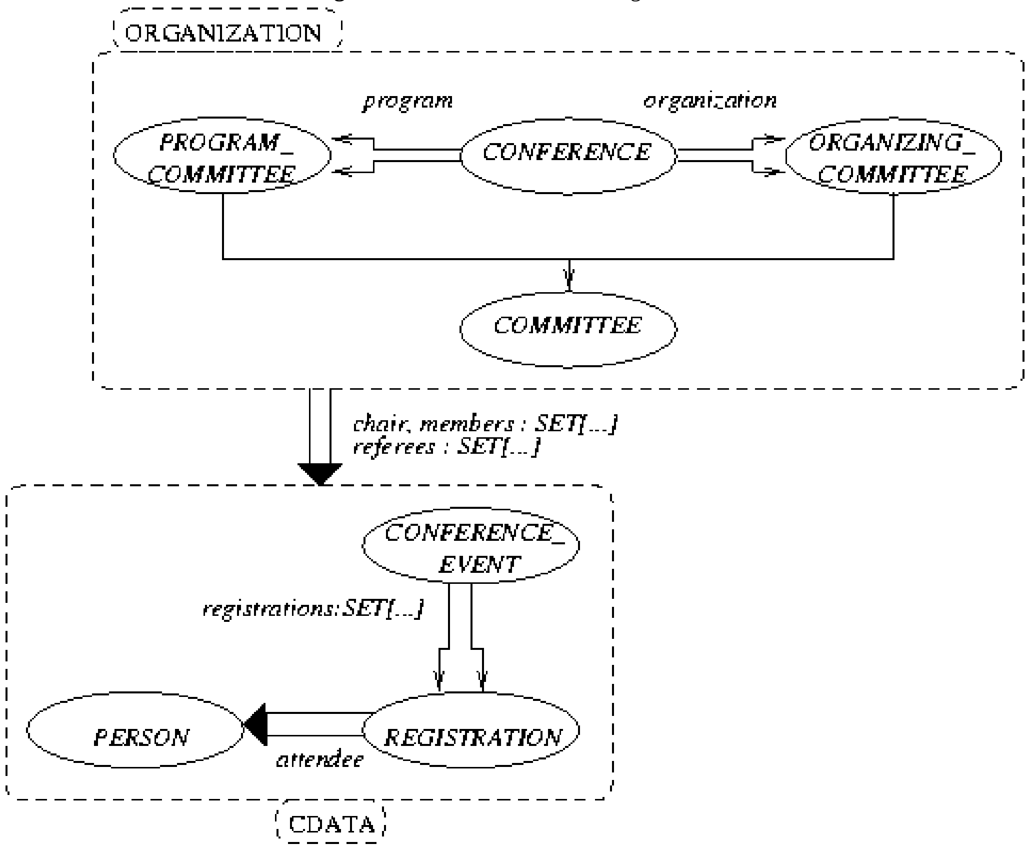
Bidirectional links appear odd: two classes cannot be part of each other! But client relations are between classes, and it is not impossible for for class *A* to have an integral part of class *B*, and vice versa, providing that different instances are involved. Think of any recursively defined data structures (e.g., trees) for an analogy.

By definition, bidirectional *aggregation* relationships are not possible.

Client-supplier and clusters

Just like inheritance, the client-supplier relation can be generalized to apply to clusters. Fig. 16 shows an example.

Figure 16: Client relations involving clusters



The *ORGANIZATION* cluster of a conference system is a client of the *CDATA* cluster. The labels refer to features of classes in the client cluster, but we cannot tell which class from the diagram. If the classes are important, then the diagram should be expanded to include such details. Client-supplier arrows can cross cluster boundaries (i.e., a class in one cluster can be client of a class in a second cluster).

Where To Learn More

There's more to BON than what we're able to show here. BON also features diagrams for describing the *dynamic* characteristics of object-oriented systems. Dynamic characteristics are the ones that arise when a system is executing. For example, classes might use the routines of other classes and may react to these messages in interesting ways, or messages might be passed between objects. We haven't discussed these ideas here; they are the topic of future courses.

If you want to learn more about BON, you should look at [1] and [3]. The former book has a concise description of BON and how it is used, but it is more a textbook on object-oriented technologies in general. The latter book is a text on BON, and contains case studies and detailed examples. You may use these books in future courses.

To learn more about other object-oriented notations, like UML, OMT, or Objectory, look in the library, or talk to your instructor.

Bibliography

1 Meyer, B. (1997) *Object-oriented Software Construction*, Second Edition, Prentice-Hall.

2 Paige, R. and Ostroff, J. (1999) A Comparison of BON and UML. In *Proc. UML'99*, Lecture Notes in Computer Science, Springer-Verlag.

3 Walden, K., and Nerson, J.-M. (1995) *Seamless Object-Oriented Software Architecture*, Prentice-Hall.

... type[1]
This is a simplification of what is going on behind-the-scenes, but for now, you can think of an object as possessing instances of the routines of its class type.

... diagram[2]
BON convention is that class names are written in all capitals.

... class[3]
Except the `creation` routines.

... invariant[4]
In fact, Eiffel relaxes this last requirement slightly and allows private routines to temporarily invalidate the invariant.