Object-Oriented Analysis and Top-Down Software Development

Dennis de Champeaux
HP-Labs
1501 Page Mill Rd, 1U
Palo Alto, CA 94304-1181
USA

Abstract

In this paper, we address the issue of how to provide an analyst that uses the object-oriented paradigm with a top-down approach. An analyst gets this approach for free when working within the structured paradigm. Ensembles are introduced that differ from objects in that they connote entities with internal parallelism. Preliminary experimentation suggests that ensembles allow for information hiding.

electronic address: champeaux@hplabs.hp.com

telephone #: (415) 857 6674

key words: OO-Analysis, top-down, ensemble

1 Introduction

In this paper, we outline a top-down object-oriented analysis (OOA) method. Top-down OOA allows an analyst to employ well-established strategies like divide-and-conquer.

We start by clarifying some of our terminology:

> *Analysis* is the activity that yields a description of *what* a target system is supposed to do; detailing functional, performance and resource requirements. This description could be the basis for a contract between the client and the developer and aims to be the unambiguous input to the designer.
>
> *Design* is the activity which yields an artifact description of *how* a target system will work. The design satisfies the requirements, while it is still implementation language independent. The artifact description aims to be the unambiguous input to the implementor.
>
> *Object-oriented analysis* describes a target system with a characterization of the entities in the domain, their inherent interrelationships, and their intended behavior in isolation as well as their interactions. The order in which these aspects are addressed varies, but usually the entity characterization precedes the behavior description. This contrasts with the order in which structured analysis deals with these aspects; behavior first and entity characterization (data dictionaries) second.

Our work is grounded on the assumption that neither structured analysis nor structured design provide a natural characterization for subsequent implementation in an object-oriented language, as supported by experience in Hewlett-Packard. At the same time, we do not suggest that object-oriented analysis and design make sense only when a subsequent implementation employs an object-oriented programming language.

The necessity of analyzing a system in a top-down fashion arises specially in the characterization of large systems. While the analysis of a toy example like the popular car cruise control system yields only a "flat" set of objects, the analysis of a corporation like Hewlett-Packard, an airline reservation system or a bank will yield "objects" at different abstraction levels.

The problem that we encounter is caused - we conjecture - by an uncritical adoption of the notion of object from the realm of the object-oriented programming languages. We suspect that this is the core reason why identifying objects is a hard task. We can wonder for instance whether the following notions are proper objects:

In the realm of Hewlett-Packard:

> a division, a department, an employee, a project, a production unit, a product, an order, a floor in a building, a location code, etc.

In the realm of an airline system:

> a flight, an airplane, a flight attendant, a client, a flight schedule, a special meal order, a service schedule, a luggage door, a payment scale, etc.

In the realm of a bank:

> an interest rate, a branch office, a teller machine, a corporate account, a loan officer, the overseas department, a monthly statement, etc.

One cannot immediately deny objecthood to any of those notions. However, their juxtaposition gives an uneasy feeling. We need different abstraction levels. The unhappy consequence is that we need to introduce objects that are "less equal", to paraphrase Orwell, than other objects. We propose *ensembles*, a different kind of abstract object, to facilitate a top-down analysis mode.

This paper is organized as follows: section 2 summarizes our current version of an object-oriented analysis method by outlining the notions for the models that the analyst can construct. In section 3, we introduce and discuss the notion of an ensemble. We illustrate ensembles in greater detail in section 4 by applying them to the example of a car, which we view as a hierarchy of multiple systems. The last section is devoted to a discussion of the pros and cons of the ensemble concept.

2 The Object-Oriented Analysis Method

Our analysis technique emerged from a variety of influences, among which are work in knowledge representation languages like KL-ONE ([2]), formal software development ([11, 5]), experiences gathered inside Hewlett-Packard at utilizing the object paradigm, and previous object-oriented analysis approaches ([9, 3, 8]). In particular, the work of Shlaer and Mellor ([9]) was the focus of our early efforts.

As mentioned above, we view the analysis process as "the activity that yields a description of *what* a target system is supposed to do by detailing functional, performance and resource requirements". The output of object-oriented analysis should satisfy two requirements:

- it should be a contract between client and developer
- it should be a contract between analyst and designer

Many approaches to OOA fail to satisfy the contract character, mostly because they fall short of providing two essential features: (1) the ability to be precise, i.e. to have a rich analysis language that allows, if desired, a rigorous and semantically unique description of the domain of discourse; (2) the provision of a development process, i.e. a framework in which a problem is composed and/ or decomposed.

Our method tries to overcome these deficiencies. It consists of the following steps:

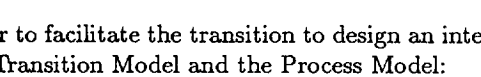
- Developing an Information Model
- Developing a State-Transition Model
- Developing a Process Model

These will be discussed in greater detail in subsequent sections (see also [4]). The reason for using these models is to provide a variety of views of an object so as to capture as much data as possible during analysis. Many of the topics related to these views go beyond the scope of this paper, for example how to find objects, how to attach identified services to particular objects, or how to migrate from OOA to OOD. The interested reader can find further information in the cited literature.

The typical sequence of model development starts with information modeling and proceeds as diagramed below:

Explanation of the symbols:
IM = Information Model
ST = State-Transition Model
PM = Process Model

In order to facilitate the transition to design an interface model, IFM, may be derived from the State-Transition Model and the Process Model:



We foresee that an interface model would generalize away the specific details of the object interactions in the process model and would produce the set of services that are associated with a prototypical instance of a class. (A service is not necessarily a synchronized interaction pair between an initiator and a recipient. Services subsume here as well trigger and send-and-forget interactions.)

2.1 Information Model

The IM consists of object class definitions, ensemble class definitions, and definitions of inter-object relations; the notion of ensemble is introduced in section 3.

Existing approaches to OOA typically define objects by listing a collection of attributes which are descriptive names (like BankAccount). There are shortcomings with this style of definition. For example, the analyst should be able to express what constitutes the legal value set of the deposit of a BankAccount. An attribute value may be dependent on the values of other attributes. Dependencies should be expressible as well. The occurrence of an attribute may be fixed or may vary over all instances of a class. It is worthwhile to register such a regularity also.

While attributes help to describe an object, we can elaborate the significance of an attribute by describing it beyond its name through its features. We have borrowed the following features from the KL-ONE knowledge representation language ([2]):

- cardinality: whether the attribute value is a singleton, enumeration, fixed or unbounded (sub)set.
- modality: whether this attribute always has a value (i.e. is mandatory), optional or whether this attribute is derived.
- value restriction: the named set of values out of which actual values have to be taken.

The analyst can also state an integrity constraint, here called *invariant* that applies to every instance of a class. Typically, an invariant is an implicitly quantified statement that refers to features of attributes. (In KL-ONE ([2]), invariants were captured by structure links; see an example in section 4, figure 1.)

Here is an example that provides attribute name, cardinality, modality, and value restriction for each attribute:
Object class BankAccount

- account owner, fixed-set, necessarily-present, Name
- account type, singleton, necessarily-present, (saving, checking)
- balance, singleton, necessarily-present, Amount
- connected_accounts, set, optional, BankAccount

An invariant for such an account would be:

$$balance + SUM(connected\_accounts.balance) > 0$$

saying that an overdraft in an account may be tolerated as long as sufficient funds are available in connected accounts.

An analyst may observe that two defined object classes have common attributes. In that case, the common attributes can be abstracted into a new object class, the common attributes can be removed from the initial classes and an inheritance relationship can be introduced between the new abstract object class and the modified classes. Inheritance can also be introduced initially as a consequence of inherent commonalities in the domain of discourse. In the banking world, we encounter checking accounts, saving accounts, commercial accounts, etc. This suggests that one introduces a generic account class and let the specific accounts inherit from it.

The graph constructed by taking the objects as vertices and the inheritance links as the arcs is directed and acyclic. This graph turns into a tree if no object inherits from multiple parents.

2.2 State Model

While the Information Model addresses the static aspects of an object, the dynamic (or behavioral) aspects are described in a State Model (SM).

The states of an object are derived from the set of all possible values of its attributes. A state is defined by a predicate on the state space spanned by the cartesian product of the value restrictions of the attributes. The predicates should be defined such that the states are mutually disjoint.

A transition corresponds with a directed pair of states. The set of states and transitions form a directed graph which is not necessarily connected. In case we have more than one component, we consider the components as independent. While an object occupies a state within each component, inside a component only one state at the time is visited.

OOA does not associate actions with states, as is done in [9], but with transitions. The state-transition model in [9] can be phrased as "states cause each other", while our method captures "transitions cause each other".

A transition carries a *condition* that is to be fulfilled before a transition can take place; that is, being in a state does not automatically enable a transition; such a condition can refer to attributes of "other" objects.

We augment the concept of state transitions by attaching:

- an *external* flag that indicates whether a triggering event is required;
- a *cause* list that describes the events that are generated as a consequence of the transition and act as triggers for subsequent transitions, usually in other objects.

In order to create objects that are reusable, we describe the dynamic dimension of an object ***independently*** of how it will interact with other objects in the context of the target system. This entails that a reference to an external object - to describe a causal consequence of a transition - should abstract away from the actual connections that the object has when integrated in the target system. To obtain proper generality, one may have to introduce attributes in an object whose role is to capture interaction "acquaintanceships" with peer objects.

As an example consider the domain of pipes, valves, junctions, pressure regulators, reservoirs, etc. In order to model the propagation of a pressure change in a pipe, we need to refer to a transition of an attached device. Since a generic pipe can't know what device it is attached to, we need a pipe attribute that stores this information.

The process model is responsible for "welding" the state models together through event descriptions.

2.3 Process Model

An analyst can express a causal connection between transitions in different objects by adding to a transition in an originator a causelist and in a recipient transition an indication that a preceding triggering event is required. In this section, we give an example to elaborate.

We describe the connection between a button object and a car cruise control object which is in a sense the "brain" of a car cruise control system. The button's responsibility is to switch the system from the off to the on state.

We model the button as a single state, single transition machine. The condition for the transition is always true. However, the transition needs a triggering event to fire. The source of this trigger is outside the system boundary and corresponds with the side effect of pushing the physical button on the dashboard. The transition has on its cause list a single event, `On-Event(ccs, turn-on)`. The `ccs` argument describes the recipient of the trigger; we assume here that the button has an acquaintance attribute `ccs`. The second argument, `turn-on`, indicates which transition in the recipient is "invited" to fire. In general, the originator has no control on whether the recipient can honor the invitation. For example, pushing the (physical) button twice should have no effect the second time.

The recipient car cruise control object will have, among others an `Off` state and a `Cruising` state, and a `turn-on` transition between the two. This transition requires an external event to fire: `On-Event`. If necessary a transition may specify an additional condition to be fulfilled. In our turn-on transition, we may require, for example, that the car has a certain minimum speed.

This description is only the tip of the iceberg. The process model can employ as well more powerful modes of interaction than just send-and-forget triggering. We may want to trigger more than one object/ transition combination and insist, for example, that these transitions are synchronized. We may want to send data along with a trigger from the originator to the recipient(s). An originator may want to obtain an acknowledgement of reception, with or

without time-outs. An originator may issue a blocking send which results in a suspension until return data is received, etc.

A recipient may be in the wrong state to honor a trigger/ send. There are different interpretations of such a situation:

An analyst has made an error; i.e. a condition has been omitted somewhere in the originator.

The trigger/ send is queued in the recipient and will be honored when the recipient arrives in the start state of the triggered transition.

The trigger/ send is lost.

Any of these interpretations can be appropriate, thus it is the analyst's responsibility to annotate a trigger/ send with the intended interpretation.

3 Ensembles

The different models that we described in the previous section allow the analyst to focus attention on different aspects of the task. The definition of the entities in the target domain is separated from the characterizations of the dynamics of the system. The description of the lifecycle of an entity is separated from the description of how an entity interacts with other entities. Still, we feel that the support for divide and conquer techniques provided by the method is insufficient. We should have the ability to acknowledge formally that certain groups of entities are tightly coupled and that these groups are entities by themselves with more or less similar features as the basic entities/ objects in the target domain. To phrase it in a more compelling manner: object-oriented analysis without an entity clustering technique is not a viable method for the characterization of large systems.

To stress the difference between clusters and basic entities, we propose *ensembles* as an alternative for objects. Ensembles share with objects the modeling apparatus that we outlined in section 2; i.e. an ensemble has attributes, has an associated state-transition machine, has the ability to interact with objects as well as with ensembles and can have an interface model. An ensemble differs from an object in that it stands for a cluster or bundle of less abstract entities which are either objects or lower level ensembles. These constituents interact only among each other or with the encompassing ensemble. I.e. the ensemble acts as a gateway/ manager between the constituents and the context. The relationship between an ensemble and its constituents can be thought of as subsuming abstract-part-of. While the dynamic dimension of an object can be conceptualized as a sequential machine, an ensemble connotes an entity with internal parallelism.

For example, in the bank domain, we can see an account as an object when only one transaction at a time is permitted on it. On the other hand, the loan department with several loan officers would be an ensemble because its constituents, the loan officers, are operating in parallel (presumably).

An ensemble hides details of its constituent objects/ sub-ensembles that are irrelevant outside the ensemble, somewhat in analogy with an object in the programming realm that hides its internal implementation details.

To make the notion of an ensemble more real, we will look in this section at its features in more detail. Section 4 discusses an example.

3.1 Ensemble Class

In the same way that we like to deal with classes of objects instead of individual objects, we will deal with classes of ensembles instead of individual ensembles. And as is the custom in the case of objects in which a class of objects is characterized with a prototypical member, called "an object", we will deal with classes of ensembles through a prototypical ensemble.

We have the following correspondences:

	descriptive notion	
target domain	atomic	cluster
entity	object	ensemble
concept	object class	ensemble class

3.2 Ensemble Constituents

Describing the constituent objects/ sub-ensembles of an ensemble is the primary task of its information model. Regular attributes can do this. An invariant relating a constituent in the value-restriction of such an attribute and the $self of the ensemble may elaborate the *abstract-part-of* relationship between the two.

Additional attributes in an ensemble may describe features that apply to the cluster of constituents as a whole. For instance, summary information of the constituents. Their number is an example. Or, as an another example, we can capture in an ensemble information that applies to each of its constituents. Consider a fleet to be represented by an ensemble. The individual ships share the direction in which they are going. Thus, we can introduce direction as an attribute of a fleet.

When an ensemble has non-constituent attributes, it may have a "life of its own"; i.e. we may develop a state-transition model for it. As an example we can maintain in a fleet an attribute that records the distance of the fleet to its home port. This allows us, for example, to introduce three states induced by a linear ordering suggested by: near-the-home-port, remote-from-the-home-port and far-away-from-the-home-port. These distinctions could have consequences in the process model for, say, refueling operations.

If an ensemble has been equipped with a state-transition model, we can describe inter-ensemble and/or ensemble - object interactions similar to the plain inter-object interactions. An example of an inter-ensemble interaction in our fleet domain, where a home-fleet is seen as the ensemble consisting of the home ports of the ships in the fleet, would be a fleet-home-docking trigger initiated by a fleet to begin the docking of the ships in the home ports. An example of an ensemble - object interaction would be the fleet giving a directive specifically to one of its ships.

4 Example

We will model fragments of a car to illustrate in greater detail the use of ensembles. A car can be seen as a single object only if one does not need to deal with its components. This would be the case, for instance, from the perspective of a car rental agency. Otherwise, when the internal aspects do matter, we better see a car as consisting of several systems, including: steering, suspension, electrical, transmission, brake, engine, heating, doors, controls, etc.

We can recognize that the entries on this list do not correspond with ordinary car attributes. They have behaviors of their own, while they operate semi independently and in parallel. In contrast, examples of regular attributes of a car are: chassis, coach-work, owner, license-number, speed, location, etc. Some of the entities on this list also have life cycles and operate semi independently, thus one may wonder why they are not constituents of a car. The justification for the different elements vary. For the chassis and coach-work one might argue that they have a state "being a part of a car assembly", which from the perspective of the car is too constant to be considered a system. For the owner, one might argue instead that the owner attribute is an artifact, the remnant of a binary Owner relationship between cars and persons that is realized ("implemented") via attributes. The other attributes refer to value restrictions which aren't objects, at least not from the perspective of a car.

In this paper, we adhere to the following graphic conventions:

```
 -----
/ XYZ \     : a trapezoid indicates an object class or ensemble class
-------

U--->       : this arrow denotes class/ ensemble inheritance

|----{...}  : a regular attribute of a class/ ensemble

|====={...}  : a constituent attribute of an ensemble

{ / / / }   : see section 2.1 for the 4-tuple inside the brackets
```

With these conventions, we obtain the following fragment of the information model of a car:

```
 -----            ---------
/ Car \  U----> / Vehicle \ ; Vehicle is a super class of Car
-------          -----------
   |-------{chassis/ 1/ np/ Chassis}
   |-------{coach-work/ 1/ np/ Coach-work}
   |-------{owner/ [1, 00)/ np/ Person U ...}
   |             ; An owner can also be a here unspecified
   |             ; non-person, more than one owner is possible
   |-------{license-number/ 1/ np/ String}
   |-------{speed/ 1/ np/ [0, max-speed]}
   |-------{location/ 1/ np/ Place}
   |========{steering-sys/ 1/ np/ Steering-sys}
   |========{suspension-sys/ 1/ np/ Suspension-sys}
   |========{electrical-sys/ 1/ np/ Electrical-sys}
   |========{transmission-sys/ 1/ np/ Transmission-sys}
   |========{brake-sys/ 1/ np/ Brake-sys}
   |========{engine-sys/ 1/ np/ Engine-sys}
   |========{heating-sys/ 1/ np/ Heating-sys}
   |========{door-sys/ 1/ np/ Door-sys}
   |========{control-sys/ 1/ np/ Control-sys}
   |========{wheel-sys/ 1/ np/ Wheel-sys}
  ...
   + structure-links:
        wheel-sys.front-wheels.angle =
              angle-function(
                 steering-sys.steering-wheel.angle)
        wheel-sys.wheel-rotation =
              wheel-rotation-function(speed)
...
```

Fig. 1 A fragment of the information model of a Car, when seen as an ensemble.

The structure-link in Car reaches inside the ensemble Front-wheels which is a constituent of the ensemble Wheel-sys. Front-wheels itself has a constituent Wheel-pair, which in turn has two Wheels as constituents. We obtain the following fragments:

```
 -----------
/ Wheel-sys \
-------------
   |-------{wheel-rotation/ 1/ np/ [0, max-rotation]}
   |========{front-wheels/ 1/ np/ Front-wheels}
   |========{rear-wheels/ 1/ np/ Rear-wheels}


 --------------
/ Front-wheels \
-----------------
   |-------{angle/ 1/ np/ [0, max-angle]}
   |========{wheel-pair/ 1/ np/ Wheel-pair}


 --------------
/ Rear-wheels  \
-----------------
   |========{differential-gear-sys/ 1/ np/ Differential-gear}
   |========{wheel-pair/ 1/ np/ Wheel-pair}


 --------------
/ Wheel-pair   \
-----------------
   |========{left-wheel/ 1/ np/ Wheel}
   |========{right-wheel/ 1/ np/ Wheel}
```

Fig. 2 A fragment of some constituents of the ensemble Car; these constituents are themselves ensembles.

Regular attributes and constituent attributes have much in common, see section 2. A constituent attribute also has a cardinality descriptor, a modality descriptor as well as a characterization of the value restriction, i.e. the kind of constituent(s) that is referred to in the attribute. For example, if we see the wheels of a Car as non-distinguished sub-constituents of the Wheel-sys constituent - unlike the modeling done above -, we can indicate that the cardinality feature is four (excluding here pathological vehicles), that the modality is necessary and that the kind is obviously Wheel. A structure link capturing an invariant can refer to a constituent attribute as well. For example, there is a constraint between the angle of the front wheels with respect to the chassis and the degree of rotation of the steering wheel as expressed above in the information model of Car.

Observe that we introduced a regular attribute wheel-rotation in Wheel-sys. A structure-link in Wheel-sys should express that the value of this attribute is the average of the rotations of the Wheels in the two constituting Wheel-pairs.

An ensemble can have a regular state-transition model (and possibly more than one, as is allowed for regular objects). For example, we can observe for our car whether it is insured or

not, whether it is for sale or not, whether the manual transition indicates neutral, rear, first, second, third, fourth gear, whether the lights are off, on park lights, dimmed or full, etc. Some of these state-models are imported from lower level constituents through the `control-sys` constituent.

As a major difference between an object and an ensemble, we have associated with an ensemble a forwarding mechanism for triggers and messages that mediates between external entities and the constituents of an ensemble. Thus, we can hide aspects of the constituents of an ensemble which have significance only inside the ensemble. For example, the interface of the engine is an internal affair of a car and the outside world need not to know anything about it. On the other hand, the forwarding mechanism of the car ensemble should export the interface of the control constituent.

We will illustrate information hiding occurring within the car ensemble by sketching the description of starting a car. We will export through the `Control-sys` constituent the state transition diagrams of an `Ignition-lock` and of an `Oil-pressure-indication-lamp`, which are both (sub) constituents of `Car`.

The state-transition diagram of the ignition lock:

`Ignition-lock:`

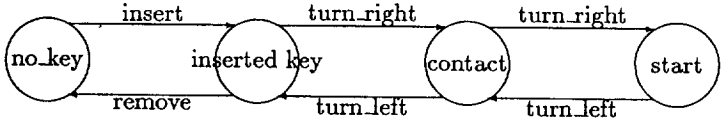


Fig. 3 The exported state-transition diagram of the ignition-lock.

The `turn-right` transition that leads into the `contact` state triggers the `contacted` transition in `Oil-pressure-indication-lamp`, see below.

Other relevant constituents that we consider are:

`Start-engine`, `Engine-sys`, and `Oil-pressure-sensor`.

Their state transitions will *not* be exported through the `Control-sys`. For the start-engine we have the following behavior description:

`Start-engine:`

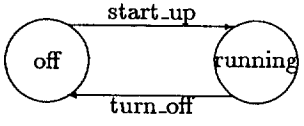


Fig. 4 The state-transition diagram of the start-engine.

We assume that the turn-right transition that connects the contact state with the start state in Ignition-lock has a trigger directed at the start-up transition in Start-engine. (The identity of the recipient object - in this example there is only one legal recipient - can be traced through the car ensemble.) We omit here conditions associated with the start-up transition, like the transmission being in neutral, etc. The start-up transition in its turn will generate a trigger aimed at the start-up transition in Engine-sys:

Engine-sys:

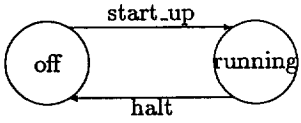


Fig. 5 The state-transition diagram of the engine.

To simplify matters, we assume that the start-up transition in Engine-sys directly triggers the go-high transition in Oil-pressure-sensor:

Oil-pressure-sensor:

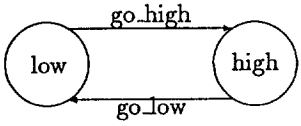


Fig. 6 The state-transition diagram of the oil-pressure-sensor

The go-high transition finally triggers the start-up transition in: Oil-pressure-indicator-lamp, which causes the lamp to go off again:

Oil-pressure-indicator-lamp:

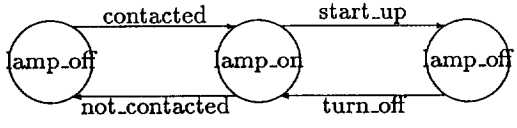


Fig. 7 The exported state-transition diagram of the oil-pressure-indicator-lamp

Since the state-transition of the `Oil-pressure-indicator-lamp` is exported the driver will see the lamp go off.

When we look from the outside, we see *pseudo* causal consequences. For instance, the `turn-right` transition out of the `inserted-key` state "causes" the `oil-pressure-indica-tor-lamp` to go on. A similar pseudo causality turns this lamp off again when the ignition-lock moves into the start state (which signals the driver to turn the key out of the start position, which causes the ignition-lock, etc.)

However, when we look inside the `Car` ensemble, we will see a different triggering/ messaging pattern that ultimately achieves these pseudo causal consequences.

In summary (and without claim to automotive correctness): starting engine → running engine → actual pressure goes up → oil pressure sensor goes in high state → oil pressure lamp goes off. Consequently, the introduction of ensembles has allowed us to successfully hide low level mechanisms from higher order functionality.

5 Related work

Object-oriented analysis is a relatively new field. The first book in this area is from Shlaer & Mellor, [9]. Most of the book is devoted to the Information Model. One chapter discusses an example in which they illustrate the State Model and their Process Model. Their Process Model differs from ours in that they rely on data flow diagrams, borrowed from Structured Analysis, to describe the actions in their State Models. As a result, the interaction between objects is described in their method in an indirect way - the occurrence of an external data store in a data flow diagram. We feel that our triggers and messages allow us to express directly causal interactions between objects. A summary of their version of Object-Oriented Analysis can be found in [10].

In Ward,[12], an attempt is made to salvage Structured Analysis and Design when an implementation will be done in an Object-Oriented programming language. Ward acknowledges that the original version of SA/SD doesn't lend itself easily to the identification of objects, and certainly not to object hierarchies which deepen the insight in the understanding of the domain. However, he points to a refinement of SA/SD for real-time systems, [13], in which entity-relationship modeling is imported from the database realm. We remain doubtful whether unbiased object identification can be done *after* processes have been modeled.

Our comment on Ward's paper, [12], applies also to that of Bailin, [1].

In Wirfs-Brock et al, [14], the authors discuss the notion of a subsystem.

> A *subsystem* is a set of ... classes (and possibly other subsystems) collaborating to fulfill a common set of responsibilities.

They motivate their subsystems similarly:

> Subsystems are a concept used to simplify a design. The complexity of a large application can be dealt with by first identifying subsystems within it, and treating those subsystems as classes.

They take an explicit position regarding whether subsystems will show up ultimately in an implementation:

Subsystems are only conceptual entities; they do not exist during execution.

On the basis of our understanding of their subsystems, we have found here the most significant difference with respect to our ensembles. Certain ensembles introduced in the analysis phase may indeed be "compiled away" in the subsequent design phase, but we foresee that at least those ensembles which have their own regular attributes in addition to constituent attributes will show up in the implementation. This explains why we felt the necessity of introducing a forwarding mechanism for triggers/ messages in ensembles. In addition, we surmise that the encapsulation provided by ensembles - constituents cannot be reached directly from outside an ensemble - is not available in their subsystems.

In the European terrain, we see two approaches as relevant for the work described here. Jacobson [6] describes a development method for large object-oriented systems, called ObjectOry, that covers the analysis phase as well as the design phase. We discuss here only the analysis component. The core notions are: entities, interface objects and use cases. Entities correspond with the objects in the target domain. Interface objects are introduced to shield the "real" objects from the system interface with the users/ external world. Use cases - as far as we understand them - correspond with generic scenarios that define the target system's behavior from the perspective of a user. (A user is to be understood in a wide sense; i.e. it can be another system.) The material that we had available did not mention (sub)systems as a way to structure a target system. Use cases, however, do provide a global view. We suspect that a use case is a special case of the information captured by a state-transition model associated with the target system represented as an ensemble.

Beta [7] is a programming language and also a development technique. The Beta language simplifies the collection of object-oriented notions by simply providing *patterns* as the only concept for classes, methods, procedures and types. As a consequence, the analysis technique reflects this simplicity, and a lot of emphasis is put on modeling the communication between objects. Beta is one of the few object-oriented systems that emphatically supports concurrency in all steps of the development process. The Beta concurrency primitives for e.g. synchronization are similar to what we have suggested for triggers and services in our method. The difference is that they have already gained experience with implementing a particular communication scheme, and that they have restricted the analysis to that scheme (ADA-like rendez-vous). In our technique, the analyst has a degree of freedom to define and use his/her own communication scheme.

6 Summary and Conclusion

Object-oriented techniques, as practiced in OOP have a bottom-up flavor since OOP does not formalize and elaborate object decomposition. This is acceptable or even desirable in the programming phase. However, an analyst needs to operate - especially in the early phase - in a top-down fashion. In this paper, we have proposed ensembles as a mechanism for clustering tightly coupled objects. This mechanism supports top-down decomposition. We have illustrated ensembles with several examples.

A major distinction between ensembles and objects is that an ensemble connotes an entity with internal parallelism, while an object connotes - from the perspective of the task domain - a finite state machine. We associate with an ensemble a trigger/ message forwarding mechanism

that mediates the interaction between external entities and the internal constituents of the ensemble. The examples discussed indicate that information hiding can be achieved indeed through ensembles.

Our ensembles resemble the sub-systems that are introduced for a similar purpose by Wirfs-Brock et al [14]. Their sub-systems appear to be a mental construct only while we foresee our ensembles to materialize ultimately in an implementation.

Experiments to validate the effectiveness of ensembles by applying the OOA method to larger real-life examples are ongoing.

Acknowledgement

George Woodmansee, Donna Ho, Penelope Faure and Teresa Parry provided illuminating feedback.

References

[1] Bailin, S.C., An Object-Oriented Requirements Specification Method, in *CACM*, vol 32, no 5, pp 608-623, 1989 May.

[2] Brachman, R.J., A Structural Paradigm for Representing Knowledge, Report 3605, BBN, 1978 May.

[3] Coad, P. & E. Yourdon, *Object-Oriented Analysis*, Yourdon Press, Prentice-Hall, 1990.

[4] de Champeaux, D., & W. Olthoff, Towards an Object-Oriented Analysis Method, *7th Annual Pacific Northwest Software Quality Conference*, pp 323-338, Portland OR, 1989.

[5] Goguen, J., Thatcher, J.W., Wagner, E.G., Wright, J.B., Initial Algebra Semantics and Continuous Algebras, *JACM*, vol 24, no 1, pp 68-75, 1977.

[6] Jacobson, I., Object-Oriented Development in an Industrial Environment, in *Proc. OOPSLA'87*, Orlando, Florida, pp 183-191, 1987 October.

[7] Kristensen, B., Madsen, O., Moller-Pedersen, B., Nygaard, K., Coroutine Sequencing in BETA, in Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, vol II Software Track, pp 396-405, 1988 January

[8] Kurtz, B., Object-Oriented Systems Analysis and Specification: A Model-Driven Approach, M.Sc. Thesis, Brigham Young University, CS Dept., 1989.

[9] Shlaer, S. & S.J. Mellor, *Object-Oriented Systems Analysis*, Yourdon Press, 1988.

[10] Shlaer, S., S.J. Mellor, D. Ohlsen, W. Hywari, The Object-Oriented Method for Analysis, in *Proceedings of the 10th Structured Development Forum (SDF-X)*, San Francisco, 1988 August.

[11] VDM Specification Language Proto-Standard, SI VDM Working Paper IST 5/50/40, 1988.

[12] Ward, P.T., How to integrate Object Orientation with Structured Analysis and Design, in *IEEE Software*, pp 74-82, 1989 March.

[13] Ward, P.T. & S.J. Mellor, *Structured Development for Real-Time Systems*, Prentice-Hall, Englewood Cliffs NJ, 1985.

[14] Wirfs-Brock, R., B. Wilkerson & L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.