Roles for composite objects in object-oriented analysis and design

Franco Civello

Computing Department,University of Brighton
Watts Building, Lewes Road, Brighton BN2 4GJ, UK.
frc@brighton.ac.uk

Abstract

A method is presented for using composite objects which separates their role and meaning as models of relations between problem-domain concepts from their role and meaning as models of hierarchical software structures. The meaning of composite objects is analysed in terms of connections between real-world concepts in object-oriented analysis and between software objects in object-oriented design. By capturing the designer's rationale for model transformation, the resulting models are easier to understand and maintain. An embedded systems example illustrates the approach.

1 Introduction

A composite object has a complex internal structure defined in terms of other objects.

A *whole-part* association (WPA) exists between the class of the composite object and the classes of each of its composing objects.

The purpose of a WPA is to describe the common properties of the whole-part links that instantiate it, just as a class describes the properties common to all its instances [Rumbaugh, Blaha, Premerlani, Eddy, Lorensen 91] .

However, classes have a well-established semantics and can be described in object-oriented programming languages, whereas there is no standardization of meaning and use for WPAs between classes ([Rubin and Goldberg 92], [Monarchi and Puhr 92]).

This paper presents a method for the definition and use of WPAs (and hence composite objects) in object-oriented analysis (OOA) and object-oriented design (OOD). The method is based on the view that WPAs are used for different purposes in OOA and OOD. In OOA, WPAs capture semantic properties of the problem-domain, whereas in OOD they capture semantic properties of the software. The nature and range of these properties and the method used for separating them is illustrated with an example of an embedded control system.

Section 2 introduces a notation and terminology for whole-part associations and composite objects. In section 3 the method is presented in outline and the motivation behind it is discussed. In section 4 the requirements of the example application are presented. Section 5 discusses the use and meaning of WPAs in OOA. Section 6 discusses the semantic properties and the use of WPAs in OOD. Finally the benefits and limitations of the method are discussed and ideas for further work are presented.

2 Notation and terminology

A *whole-part association* (WPA) is an association between two classes, the *composite* or *whole* class and the *part* class. To distinguish WPAs from other associations, the OMT convention of drawing a

OOPSLA'93, pp. 376-393

diamond shape on the association link, next to the whole class box, is adopted (Figure 1). A WPA is instantiated by a link between a composite (or whole) object and a part object.

A *whole-part structure* [Coad and Yourdon 91] includes a composite class, all of its part classes and all the WPAs between the composite and its parts. The classes Car and Engine and their WPA form a whole-part structure.

The Car-Engine WPA is mandatory in both directions, i.e., each Car must have an Engine and each Engine must be part of a Car. Other WPAs may be optional in one or both directions.

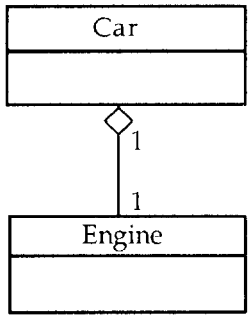

Figure 1: a whole-part association

2.1 Aggregates and Collections

Two patterns of whole-part associations recur frequently in object-oriented models. *Aggregates* [Coad and Yourdon 90] are patterns where a class has several named part classes, each with multiplicity 1 (Figure 2). The name of a part class can be omitted from the diagram if no ambiguities can arise (e.g., Text in TextBox).

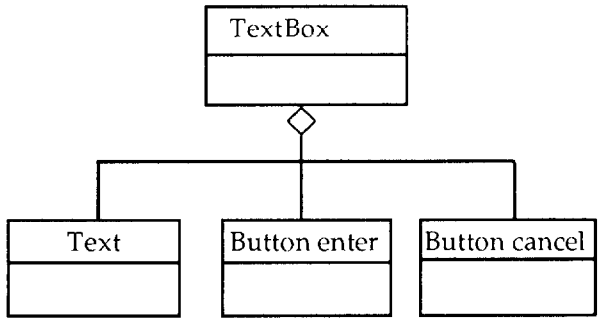

Figure 2: an aggregate

By default, a WPA is taken to be a (1) to 1 association: i.e., an instance of the whole class needs a link to one instance of the part class, whereas an instance of the part class can exist without a link to an instance of the whole class. (Each TextBox needs a Text and two Buttons, but Text and Button objects do not exist just as parts of a TextBox).

The second recurrent pattern of association is the *collection*, in which a composite object is linked to many part objects of the same class.

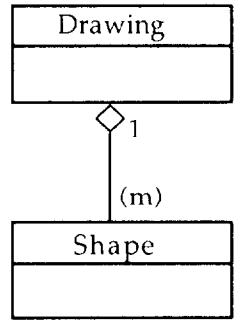

Figure 3: a collection

For example, in a graphical editor for geometrical drawings, we can model the association between the class Drawing and the class Shape, of which elements of the drawing are instances, as a collection (Figure 3).

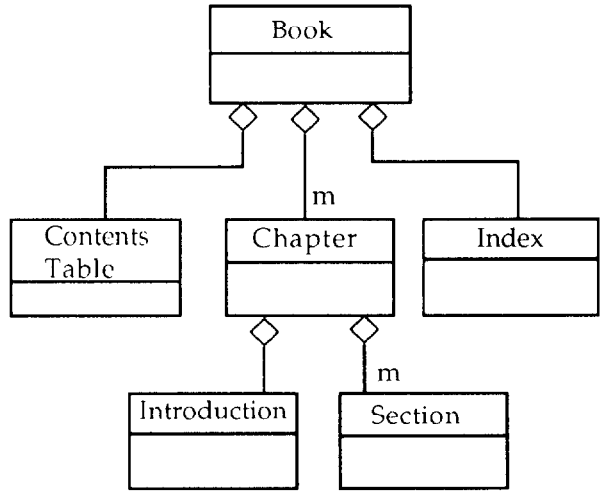

Figure 4: a multi-level composite

2.2 Composition hierarchies

To model complex hierarchies of objects, it is often necessary for a part class in a WPA to be the composite class in another. So whole-part associations can induce multi-level object composition hierarchies (part-of hierarchies) (Figure 4).

3 Rationale and outline of the method

WPAs can be used to model "part of" relationships between entities in a domain (e.g., OMT [Rumbaugh et al. 91], OOA [Coad and Yourdon 91]) and to control design complexity by encapsulating the parts of composite objects (e.g., OOAD [Booch 91], HOOD [Robinson 92], [deChampeaux 91][1]). These two goals are difficult to separate by looking at a finished model, as this not only attempts to reflect the structure of a problem domain, but is also "designed" to be understandable, manageable, reusable, resilient to change and to result in software with desired computational features, such as performance and physical distribution. This situation arises from the twofold purpose of object-oriented models: to describe the structure and behaviour of entities in the problem domain (analysis), and to describe the structure and behaviour of the software components of the system (design). During OOD an object model is refined and transformed to address design issues that are not considered in analysis [deChampeaux, Lea and Faure 92]. New WPAs can be created. Existing ones are viewed from a new, software perspective and can therefore change their properties or acquire new ones.

When a model undergoes substantial transformation during design, it is vital that the analysis model, reflecting the client's and analyst's understanding of the problem domain and system requirements, is preserved. The transformation steps must also be recorded. This way, when the requirements change or are extended, the analysis model can be modified and the transformations checked for consistency with the new requirements. If the design model is modified directly, problem and solution domain issues cannot be distinguished and considered separately. This makes system evolution harder to control and more error-prone.

It follows that WPAs must be documented so that their analysis and design properties, and the different constraints and decisions they reflect, can be separately identified.

Furthermore, within their analysis and/or design roles, WPAs can be used for multiple and different purposes which affect their semantics. This view is supported by research in cognitive psychology [Winston et al 87], which has shown that there are different types of part-whole relations (meronymic relations) between concepts, with different semantic connotations. For example, the whole-part relations Person-Arm and Company-Person are semantically different[2].

This variety of purposes and semantics is not supported by the notations of current object-oriented methods, which tend to bury it under a single notational construct, ending up with a concept too broad in scope to have a precise meaning or a useful role within the development process. Table 1 summarizes the terminology and approach of a representative sample of current object-oriented methods.

A striking feature is that those methods that use WPAs in the analysis stage (e.g., OMT, OOA) do not distinguish between WPAs and other class associations in design and implementation, whereas those methods that use them for software design purposes (e.g., Booch, HOOD) do not extend into the analysis stage. Thus no existing method gives rules or guidelines for using WPAs throughout analysis, design and implementation. In fact, the difference between WPAs and other associations is often only cosmetic and diagrammatic. While it is generally acknowledged that whole-part associations bind

[1]deChampeaux (91) uses the term *ensemble* to refer to special kinds of composite objects that encapsulate their components. This concept is subsumed by the categorisation of composite objects given in this paper, where encapsulation is only one of the design roles of a composite.

[2]The transitivity property (i.e., if A is part of B and B is part of C, then A is part of C) is lost when relations with different semantic properties are involved (my Arm is *not* part of my Company).

classes more strongly than other associations, there are no further rules or constraints to guide design and implementation decisions. For example, the duties of a composite object as owner and manager of its parts are not sufficiently elaborated by any existing method, although the TROLL language [Hartmann, Jungclaus and Saake 92] allows the representation of structural and behavioural connections between a composite and its parts.

This paper proposes a method for using and documenting whole-part associations throughout model development, that addresses the problems mentioned above:

• In OOA, WPAs are shown individually alongside other associations and are named explicitly in precise domain-specific terms, rather than in generic terms such as 'is part of' or 'includes'. In addition, each aggregate and collection is classified as belonging to one of three semantic patterns and textually annotated accordingly. This requires the developer to invest more resources in understanding the domain better, but it pays dividends in later stages by making the model easier to comprehend and providing more precise guidance for the designer.

• In OOD, more emphasis is placed on the object composition hierarchy rather than individual associations. The design properties of each whole-part structure are captured using annotations. A rationale is given for each composite object in the object composition hierarchy by cross-referencing corresponding WPAs in the analysis model and/or by stating its design purpose within the model. This gives traceability of WPAs by documenting the rationale for the transformation between the analysis and the design model.

	OOA	OOAD	OMT	CRC	Objectory	HOOD
Source	[Coad and Yourdon 91]	[Booch 91]	[Rumbaugh et al. 91]	[Wirfs-Brock et al. 90]	[Jacobson 92]	[Robinson 92]
Notation support for WPAs	Yes	Yes	Yes	No	No	Yes
Terminology	whole-part structures	'has' relationship	aggregation (whole-part association)	'part of' relationship	'consists-of' relationship	'include' relationship
Problem-domain modelling semantics	strong (pervasive organising principle)	weak	strong (parts' common properties)	weak	weak	none
Variations	container collection assembly	by-value by-reference	none	container composite	none	none
Definable properties	optionality multiplicity	optionality multiplicity	optionality multiplicity	none	none	none
Software design semantics	none	ownership/ encapsulation	none	none	none	encapsulation /delegation
Relevance in method	fair	major	minor	minor	minor	major
Main purpose	mental aid in analysis/ subsystem identification	abstraction/ decomposition/hiding	problem-domain modelling	mental aid in design	mental aid in modelling	decomposition/ design/ distribution

Table 1: summary of use of WPAs in representative methods

4 The example system

The system specification is derived from a published example of real-time structured design techniques [Ward and Mellor 85]. The system consists of a number of bottle-filling lines fed by a single vat containing the liquid to be bottled. Figure 5 shows some of the details of the vat apparatus and of a representative bottling line. Because of the single vat, the composition of the liquid being placed in the bottles is the same for all lines at a given time. However, the bottle size may differ from line to line.

The tasks of the control system are to control the level and the pH of the liquid in the vat, to manage the movement and filling of bottles on the various lines, and to exchange information with human operators working the individual lines and with a supervisor monitoring the entire system.

The vat level control is accomplished by monitoring the level with a sensor and adjusting a liquid input valve accordingly. The requirement for controlling pH arises because the liquid to be bottled reacts with its surroundings, causing the pH to "creep" over time. A constant pH is maintained by introducing, through the pH control valve, small quantities of a chemical that reverses the pH "creep".

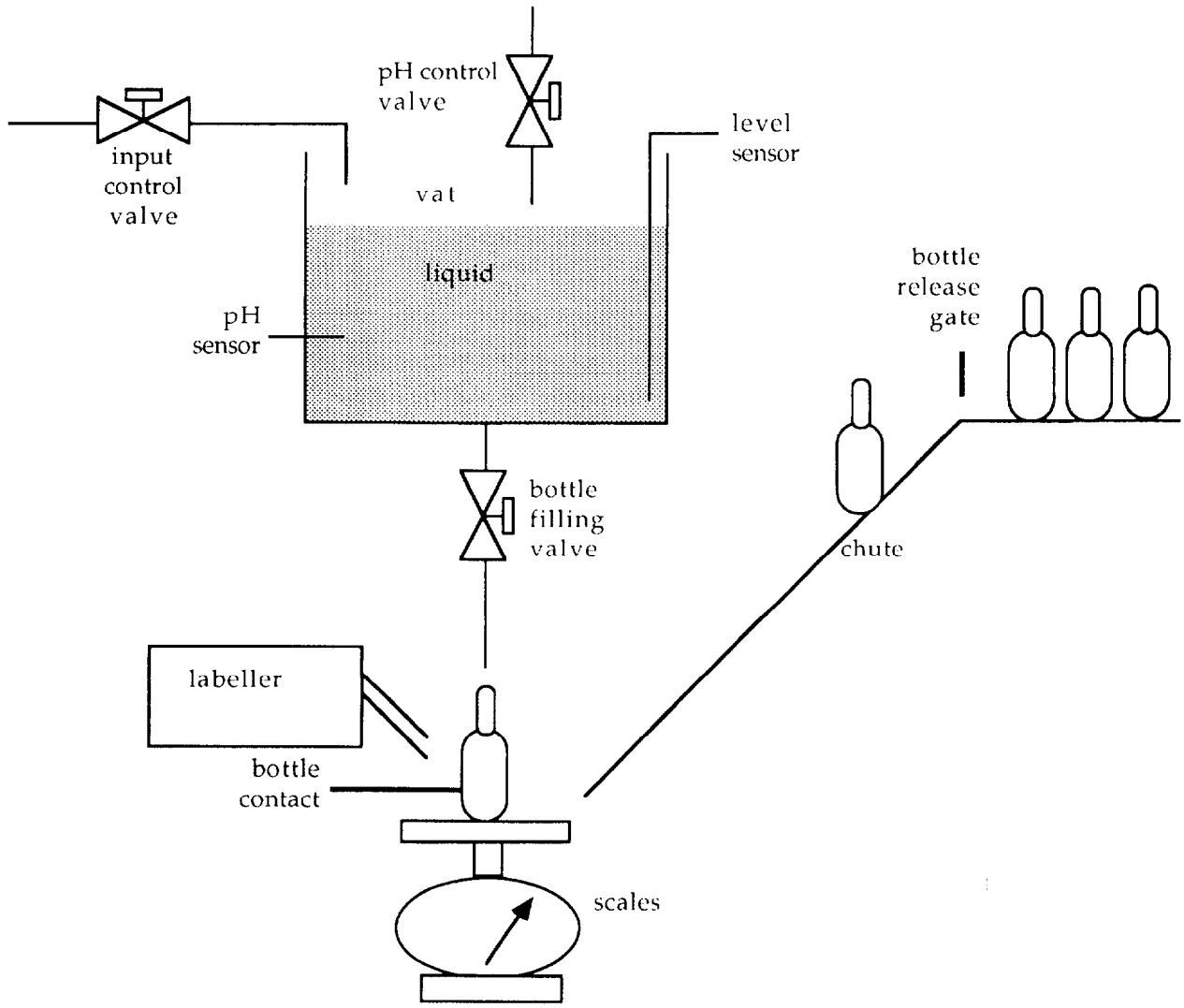

Figure 5: the physical structure of the bottle filling plant, excluding the supervisor and the operator interfaces. Only one bottle filling line is shown.

Bottles to be filled on a particular line are drawn one by one from a supply of bottles, as follows:

- A bottle is released from a gate and drops down a chute onto a scale platform, at the same time depressing a bottle contact sensor. The bottle is weighed empty.
- The bottle-filling valve is opened, and a measured amount of liquid is let into the bottle. The weight of the bottle plus its contents is used to determine when the bottle is full and to shut off the valve.
- The filled bottle is labelled to show the actual pH when filled, and the nominal pH. The line operator caps and removes the filled bottle, and signals the systems that the bottle has been removed. Removing the bottle releases the bottle contact sensor, removes the weight on the scale and allows the next bottle to be released from the gate.

The line operators can signal the system to start and stop individual lines, and the supervisor can signal the system to enable or disable the operation of all the lines. The line operators are given displays of the line status and are able to change bottle size for the line. The area supervisor is given a display of the current status of the system pH, vat liquid level and statuses of the individual lines, and is able to control the pH of the bottled liquid by entering a new desired pH to be maintained.

If, during operation of the system, the pH goes out of limits (>0.3 from the setpoint) all control actions are suspended. The vat pH is then stabilised manually. When the pH is back within limits, the system restarts automatically

5 Whole-part associations in the analysis model

Class associations identified during analysis model connections between objects in the problem domain (Figure 6). The key class attributes in the model are shown inside their class symbol.

The associations provide a basis from which to derive the dynamic communication links amongst software objects, although they do not prescribe the directions of the links, nor their implementation mechanisms[3].

The WPAs are described with domain-specific terms (e.g., Gate *releases bottles on* BottlingLine) rather than generic ones (e.g., Gate *is part of* BottlingLine), to convey more precisely the role of the links.

5.1 Semantic patterns of object composition

WPAs can be divided into two categories: *functional* and *non-functional*. In a functional WPA the part is conceptually included in the whole because of structural and functional connections that make it possible for it to contribute to the function of the whole [Winston et al. 87]. For example, the devices which make up a BottlingLine are structurally situated and connected in such a way to support the function of the BottlingLine (e.g., the gate is connected via the chute to the platform to which the contact sensor is attached). Each part object has a function to fulfil that contributes to the function of the whole object. We call the parts in a functional WPA *components* to emphasize their essential role in the association. We call the whole object an *assembly*, although no physical existence is implied. Essential parts of physical systems (e.g., the engine in a car), organisations (e.g., the headteacher of a school), or conceptual entities (e.g., the activities in a project plan) fall within this definition.

Non-functional WPAs model looser connections between whole and part. Such WPAs can be divided in two categories: *tuple-element*, and *group-member*, corresponding to the notions of *aggregation* and *association* relations in Semantic Data Modelling

[3]Additional textual constraints are required to capture the full semantics of the associations. For example, it should be stated that the set of BottlingLine objects linked to the Vat's Supervisor, found following the path 'Vat->Supervisor->BottlingLine', is the same set of BottlingLine objects found following the path 'Vat->Valve->BottlingLine'. In other words the Supervisor supervises all the bottling lines which are fed by the filling valves, and no others.

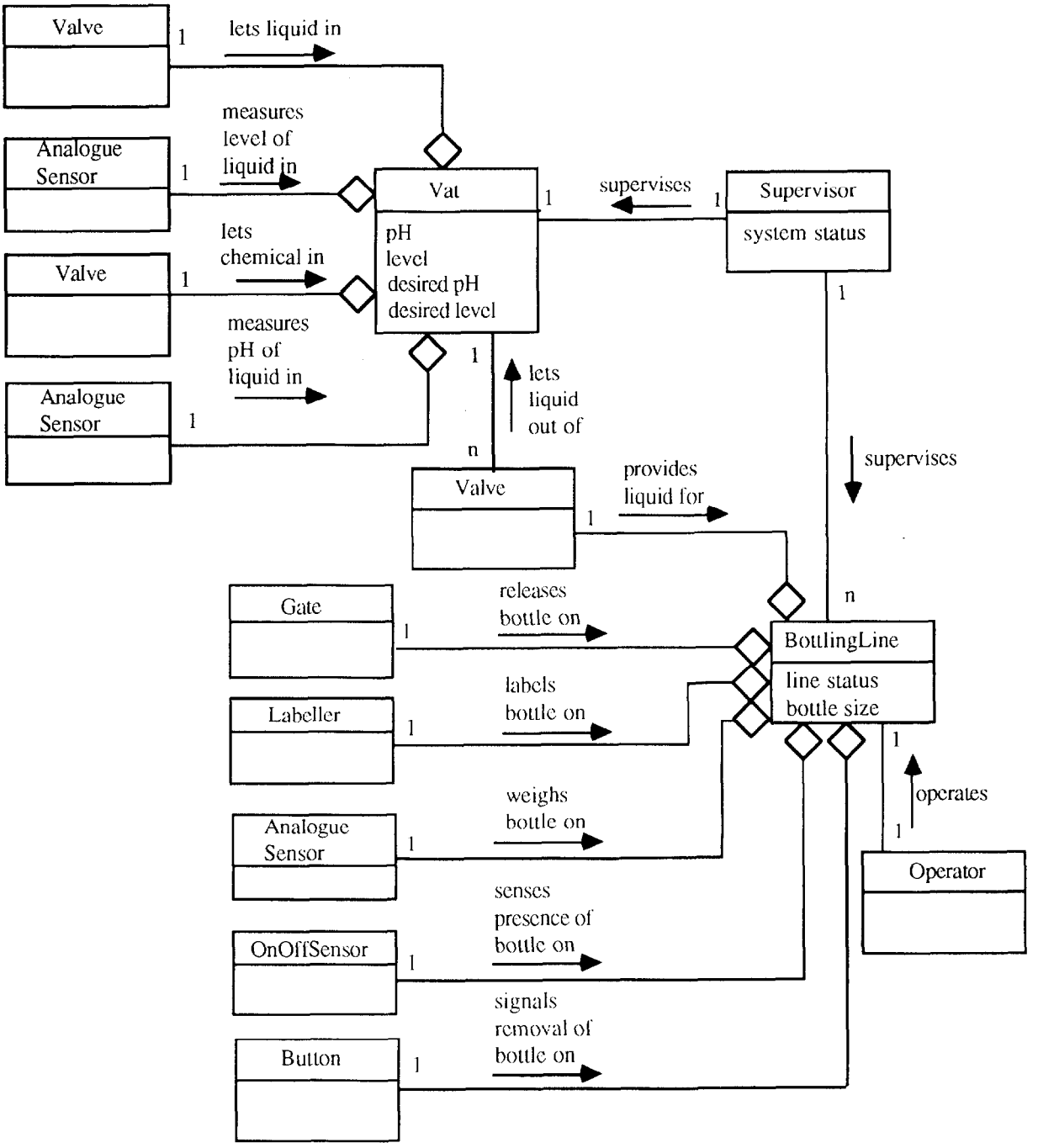

Figure 6: Class associations in the Bottle Filling System

[Hull and King 87]. A tuple models a relation between two or more entities.The two entities normally exist independently from each other. Tuples are aggregates where the names of the elements convey the roles they play in the relation (e.g., Marriage(Husband, Wife), Sale(Purchaser, Vendor, Property), Registration(Car, Owner)). Often the tuple object models an event and its elements model the entities participating in the event.

Groups are sets of objects brought together by virtue of sharing some property or by some other looser connection. Examples of group-member associations are: Hotel-Room, Committe-Member, Document-Page. In the bottle filling plant, should we

require to keep track of each bottle filled, the set of filled bottles would be modelled as a group object with no functional relation between whole and parts and no structural connections between the member objects (FilledBottles-Bottle).

Just as aggregates and collections provide *syntactic* patterns of object composition, so assembly-component, tuple-element and group-member provide *semantic* patterns, since they capture the purpose for which WPAs are being used.

Components in an assembly normally appear as the named parts in an aggregate pattern, as each has the ability to fulfil a different function, described by its name, within the composite. Members in a group, on the other hand, tend to be parts of a collection. They are not individually named because they all play a similar role from the viewpoint of their composite and none of them individually is essential to the function of the composite object.

So functional WPAs describe stronger, usually non-optional, links than non-functional ones. Each composition pattern should be annotated to reflect this distinction and to justify it. In particular, for each group object the analyst needs to state the common properties that brings together all member objects. For each assembly, the function of the composite and the contribution of each component must also be described.

In the Bottle Filling System (BFS) all WPAs are functional and non-optional. An example of a model annotation is given below:

Whole:	*BottlingLine*
Parts:	*Valve fillingValve, Gate, Labeller, AnalogueSensor scales, Button removeSignal, OnOffSensor contactSensor.*
Kind:	*Assembly*
Function:	*Manages the movement and filling of bottles on a single line. It controls the actions and monitors the state of each component device.*

In summary, WPAs are useful to highlight strong associations in a problem-domain, but come in different flavours and need to be described accordingly. A designer would be entitled to ignore the difference between WPAs and other associations if their meaning was not made clear.

5.2 Other (non-WP) associations

This method of classifying WPAs helps to tell them apart from other associations, which are sometimes confused with WPAs :

Spatial or temporal inclusion (e.g., Room-Desk, Process-Phase). Spatial (temporal) inclusion or proximity is a good heuristic clue to identify a WPA. However it does not justify a WPA in the absence of structural or functional connections relevant to the system responsibility. The filling Valves, for example, may by physically closer to the Vat than to the BottlingLines, but they are functionally closer to the latter, as their operation depends on events happening within them.

Spatial (temporal) inclusion justifies the sharing of some spatial (temporal) properties between objects, so it might form the basis for a group-member WPA, but it is semantically weaker than a functional WPA. Thus the spatial features of the Pilot-Aircraft association [Coad & Yourdon 91] do not justify a WPA. Pilots and Aircrafts are independent entities. One just happens to be inside the other for a time period. Their association is semantically similar to that between a remotely-controlled aircraft and its human controller.

Attribution (e.g., Building-Height). Height is not part of a building, but one of its attributes. Attribution is often confused with whole-part because the distinction between an attribute and a part of an object is often lost in an object-oriented implementation. For example the height of the building and the heating-system in the building would both be implemented as instance variables of class Building in Smalltalk. Furthermore, in object-oriented modelling the choice between attribution or WPA can be subjective as well as purpose and context dependent. Are the start-point and end-point attributes or parts of a line segment? The answer depends on the context: are the delimiting points used just as information holders, to store and provide access to their coordinates, or do they have behaviour that can be invoked by their LineSegment or other objects? In the latter case the

two points should be modelled as part objects. Since such behavioural decisions are often taken during OOD, an attribute in an analysis model can become a part object during design.

Class membership (e.g., John Smith - Person). This is a relation between an instance and its class, not between two instances. Semantically it expresses the fact that the properties of John Smith are defined by the class Person. However if one takes an extensive view of meaning for classes (a class is a set), it is tempting to treat a class as a collection of all its instances. Although possible, this is bad practice at both the conceptual and the practical level. At the conceptual level, it confuses the member-collection relation, based on the connections between or the extrinsic properties of a group of objects, with the class membership relation which is based on the intrinsic properties of the class members. At the practical level it creates a computational abstraction with two distinct responsibilities: defining and creating instances of a class, and keeping and managing the instances of the class. The latter is usually application and context dependent whereas the first is fixed. In addition, there is often a need for distinct collections of objects of the same class in an application. The extensive approach creates a displeasing asymmetry between how different collections of objects of the same class are handled.

6 Whole-part associations in the design model

The WPAs in the design model are shown as an object composition hierarchy (Figure 7) to emphasize that their main role is in structuring the software system as opposed to modelling the problem domain.

Whereas a problem-domain association captures, in application-related terms, the purpose for which objects are linked, composite objects place objects (and hence their classes) in a logical hierarchy, so that software can be designed in layers of abstraction, with functional responsibilities suitably distributed among the layers.

Some WPAs in the object composition hierarchy do correspond to problem-domain class associations, in which case the relevant links are replicated across the two diagrams (e.g., BottlingLine - Gate). Others do not have such semantic support in the class association model, but are introduced to make the model more suitable to a software realisation (have a design role only).

6.1 Semantic properties of WPAs in a design model

In OOD, a WPA models a part-of relation between software objects, not between real-world entities or concepts. Therefore, its semantics should be based on properties of software links.

However it is counter-productive to give necessary and sufficient conditions for calling a software link a whole-part link. The resulting conditions are either too prescriptive or too broad, and therefore unhelpful. The reason is that the concept of whole-part association in software has too many facets and shades and so defies excessive simplification.

It is more fruitful to consider the primitive properties of each WPA and annotate the model accordingly. This way the designer is free to use WPAs as he or she sees appropriate, provided some minimal necessary conditions are satisfied, but is also forced to define what is meant by each WPA.

The list below is an attempt to establish the primitive semantic properties of software links on which WPAs are based.

Visibility. A necessary (but not sufficient) condition for an object to be part of another is that the whole object has the ability to send messages to the part. Thus the composite class is a client of the part class. The converse may also be true if the application requires it. In MacApp and other GUIs, for example, each View holds a reference to its enclosing View in order to propagate events.

Encapsulation. An encapsulated (or nested) object is only visible within the scope of its encapsulating object. A composite object may encapsulate its parts, making its internal structure invisible to its clients. Current programming languages do not fully support encapsulation, as a private instance variable can be assigned to a method argument, making the part object visible outside the whole object. Component objects should be

encapsulated by their assembly to separate the external functionality of the assembly from its internal structure and functions, just as in real life complex artefacts present a simple external interface that shields the user from their internal workings. Elements of a tuple are not usually encapsulated by the tuple, as their function within a system is not just subsidiary to the tuple.

Encapsulation can be further constrained or relaxed by limiting or extending the visibility within and across the composite object:

Inward Visibility. A client of the encapsulating object can use the encapsulated object, but only by obtaining a dynamic (i.e., released after method completion) reference to the latter, from the former, during execution of one of the client's methods. This is similar to Hogg's (91) *islands*, with the whole object playing the role of *bridge*. Islands limit the scope in which an object can be statically aliased, making a design more amenable to proofs of correctness [Hogg 91].

Outward visibility. The encapsulated object may be granted static or dynamic visibility to objects outside the scope of the encapsulating object. In the Bottle Filling System, for example, a BottlingLine has visibility to the Vat, to obtain pH data.

Inward and/or outward visibility arise from associations between a part class and classes outside its composite (e.g. through its Valve, BottlingLine has an association with Vat (Figure 6)).

Whole-independence. A whole-independent part has no visibility to its whole.

Peer-independence. A peer-independent part has no visibility to other parts of the same whole.

Separate part. A part that is both peer- and whole-independent. Such an object depends only on its own parts, if any. An object composition hierarchy where all parts are separate induces a strictly hierachical interaction scheme, in which every sub-tree of the object composition hierarchy is totally self-contained. Strict hierarchies enhance robustness of designs at the expense of flexibility.

Sharing. An object is shared if two or more objects hold references to it. A part object can be shared by multiple composites (e.g., a programmer can be a member of a development team and of a quality review group).

A shared part object cannot be encapsulated, as it must be visible to more than one composite.

Part-Whole Inseparability. A separable part can be disconnected from its whole. An inseparable part cannot: its existence depends on the existence of a connected whole. For example each filling Valve is inseparable from its FillingStation (Figure 7). A separable part can be created by some other object and subsequently *acquired by* the whole; or *released* from its whole and passed on to another object. For example the messages in a mailbox are produced somewhere else and inserted into (acquired by) the mailbox. Later they will be released to be used and kept or deleted by some consumer object.

Whole-Part Inseparability. The existence of the whole object may depend on the existence of the part object. An inseparable whole will create or import its part at creation time. The part object cannot be deleted without causing the deletion of the whole. For example, an OperatedLine depends on the existence of a BottlingLine and an Operator (Figure 7).

Inseparability is therefore about the relation between the objects' lifetime. If a part is inseparable from its whole, then its lifetime is included in that of the whole object (Figure 8). Conversely, if a whole is inseparable from its part, then the lifetime of the whole is included within that of its part (Figure 9). Mutual inseparability (part-whole and whole-part) means that the two lifetimes coincide (Figure 10). It is usual for assemblies and their components to be mutually inseparable, for tuple objects to be inseparable from their elements, and for members in a group to be separable from their whole.

Together, the properties of inseparability and encapsulation correspond to *ownership*, or *has-by-value* relationship in Booch (91). Keeping the two properties separate provides greater modelling

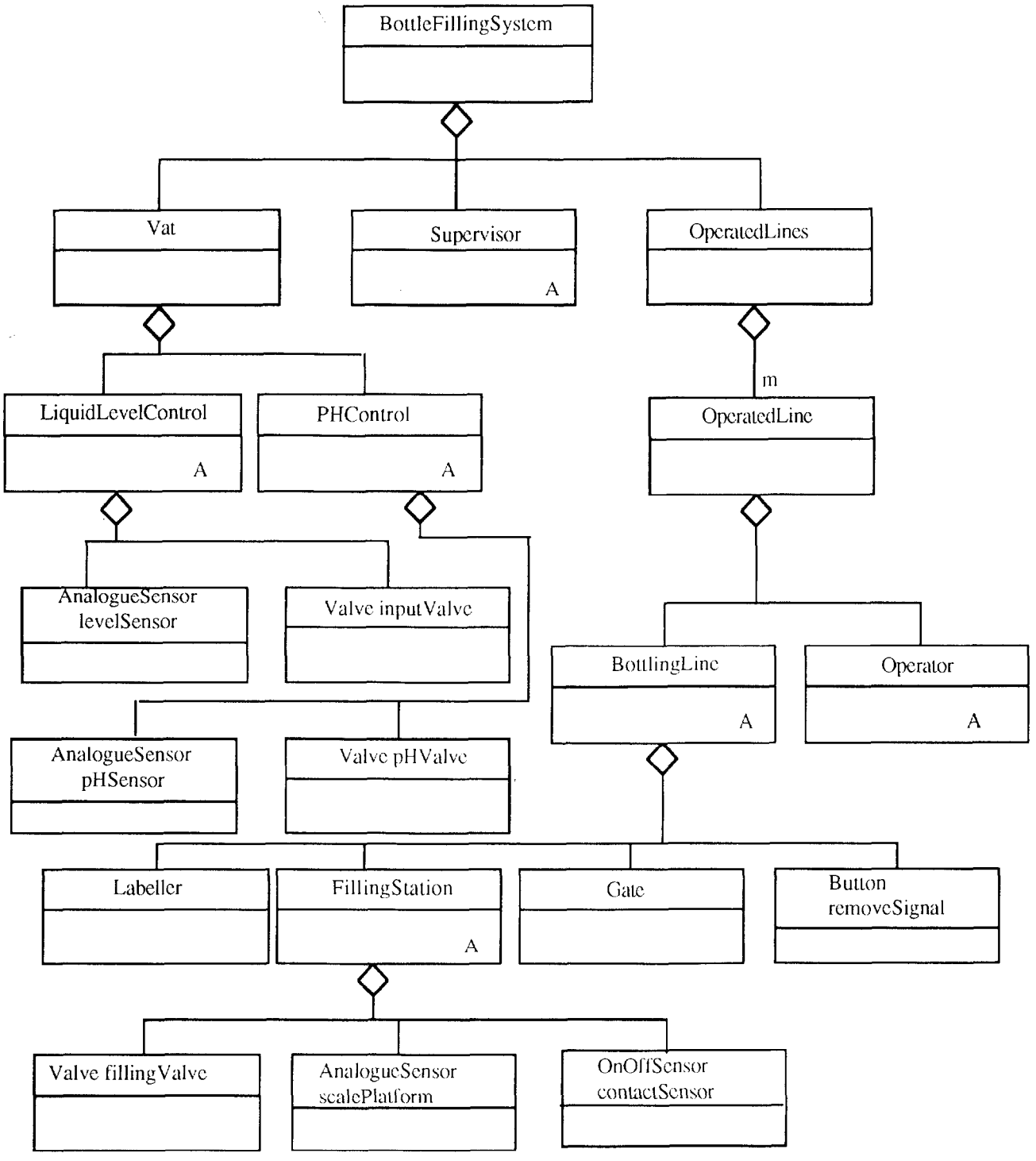

Figure 7: Composite object hierarchy in the Bottle Filling System

flexibility (e.g., inseparability with no encapsulation).

Immutability. In an immutable WPA the identity of the part object cannot change [Odell 92]. For example, in a Marriage, the identity of the husband or wife cannot be changed. If it did, a different Marriage object would be involved. Thus the entity modelled by the whole object would no longer be the same entity if one of its parts changed. In an immutable WPA, the part is separable from the whole, but the whole is not separable from the part. An inseparable part cannot be

mutable, but an inseparable whole may have a mutable part. A sailing boat, for example, needs a sail (inseparability), but the sail can be changed for an equivalent sail without affecting the function of the boat. Thus it is the *role* of the part that is essential, but not its *identity*.

Ownership. Ownership and encapsulation of a software object are treated as synonyms by some (e.g., Booch (91), Atkinson (92)). Instead, we define ownership in terms of the way that the destiny of the whole and part objects are interlinked. More precisely, an object owns another if deletion of the whole object implies deletion of the part object. This definition of ownership is a pragmatic one: it allows us to represent situations where an object is owned but not encapsulated by another object and where creation and deletion of the same object are carried out by different objects (this is quite common with objects that represent dynamic real-world objects that undergo a series of processes before coming to the end of their life). Also note that ownership is weaker than inseparability: for example a member object owned by a group may be owned by the group but also separable from it (it can be released and continue its existence outside the group).

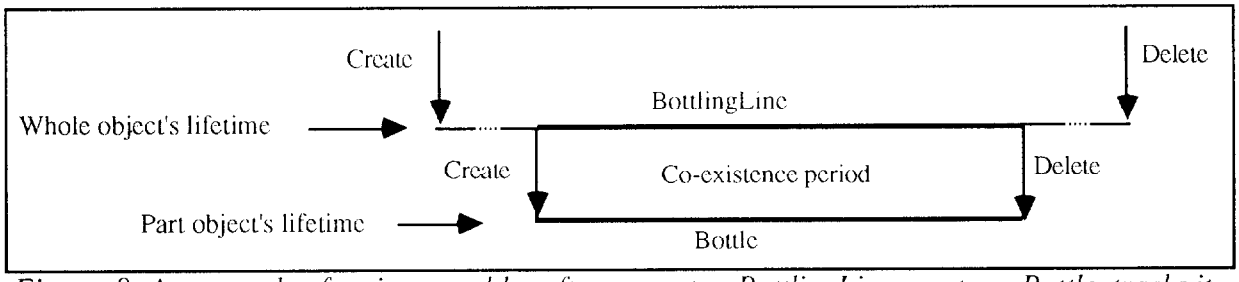

Figure 8: An example of an inseparable software part: a BottlingLine creates a Bottle, tracks it and destroys it when it leaves the line. In real life the bottle is not inseparable from the line.

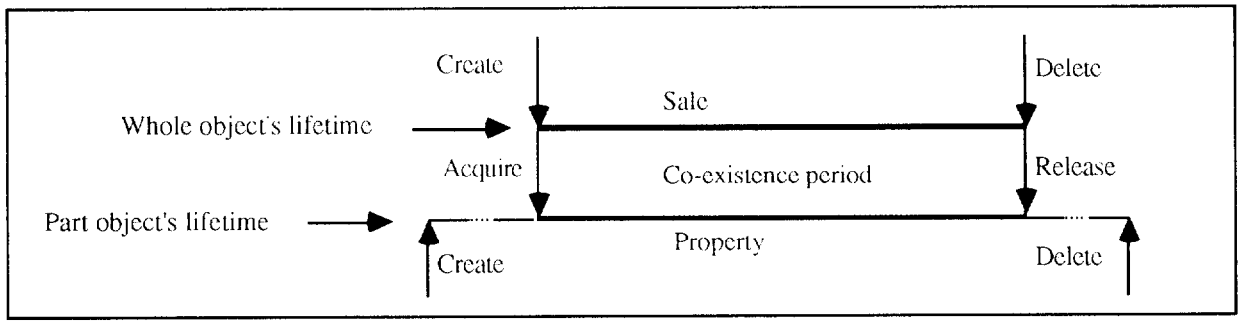

Figure 9: An example of an inseparable software whole: a Sale cannot exist without a Property. The part object in this case precedes and survives the whole object.

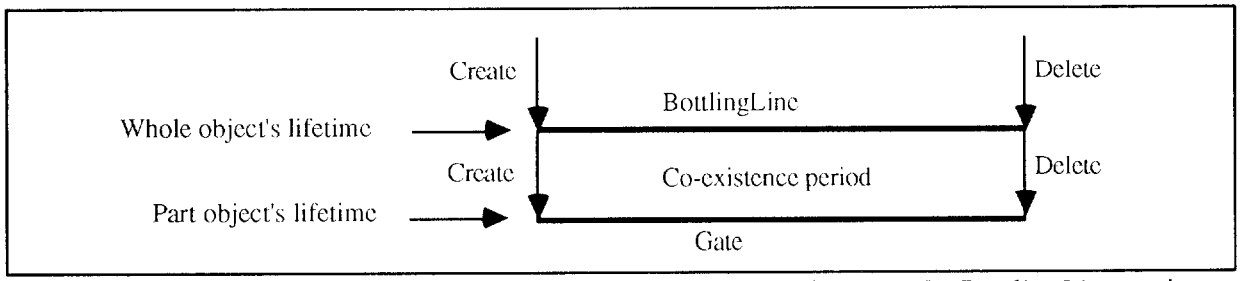

Figure 10: An example of a mutually inseparable WPA: the lifetimes of a BottlingLine and a Gate coincide.

Collaborations. Although in some composite objects (called containers by Wirfs-Brock et al. (90)) the whole objects do not call operations on their part objects nor viceversa, normally, strong collaborations exist between a whole and its parts. The nature of such collaborations is application dependent, but a few general categories can be identified.

Constraint Maintenance: where a constraint must hold that involves all or some of the parts, the composite object can take charge of ensuring that the constraint is satisfied. A special case of collaboration arising from the need to maintain a constraint is *propagation* [Rumbaugh et al 91], which occurs when the value of an attribute or link is shared between the whole and its parts. Changes to the value must be propagated or *broadcast* to each part object.

Configuration. A composite object can be responsible for configuring its part objects. We distinguish *internal* from *external* configuration. The former involves binding an object to other objects in the system; the latter sets up a link between an object and an interacting entity in the system environment.

Internal Configuration. Part objects often collaborate with their peers, and, sometimes, with clients or servers of their whole. A whole object is ideally placed to set up such links, as it provides the context within which its parts operate. Internal configuration of part objects by their composite objects makes the part objects context independent and therefore more reusable [Kramer, Magee, Sloman and Dulay 92].

External Configuration. Interface objects modelling entities in the physical system environment that interact directly with the system need to be externally configured. If the physical interfaces are arranged into structures or sets corresponding to the whole-part structures in the object model, then it is convenient for composite objects to set up the links between their parts and their physical counterparts.

Delegation of active behaviour. Objects can be passive or active. An active object has its own execution thread. Active objects are denoted by an "A" in the lower right corner of their icon (Figure 7). For the sake of conceptual simplicity, an object can have at most one execution thread [Kramer et al. 92]. However, composite objects, whether passive or active, may include active parts. So complex dynamic behaviour within an object can be decomposed by delegating part of it to the object's parts.

A simple and easily verifiable case of algorithmic decomposition arises where a complex state in the state chart of an object (i.e., a state with an internal activity that can itself be represented as a state chart) is transformed into a component object.

Control. The dynamic behavior of an object can be modelled as a finite state machine. *States* are abstractions of the values and links held by an object, and represent its dispositional behaviours: in different states an object reacts differently to the same event. *Transitions* between *states* are caused by *events* generated by other objects or by events external to the model. See Coleman (91) for how to use *object charts*, an extension to state charts [Harel 87], to model dynamic object behaviour.

An object *controls* another if it generates events for it (i.e. sends it messages that fire transitions between states). In principle events can be generated across any object link. However the complexity of object interactions, and with it the potential for data corruption, race conditions or deadlock, is reduced if objects do not mutually control each other and if control links are kept to a mimimum and explicitly documented in a model.

The object composition hierarchy can be used for the purpose of reducing behavioral complexity by giving composite objects the role of sole controllers of their active parts. This should not be considered a rigid rule but only a flexible guideline to be applied as long as it does not distort the correspondence between the model and the problem-domain.

In the Bottle Filling System, for example, each composite object is the sole controller of its active parts (there are no shared active parts), except in two cases where the control relationships are already clear in the problem domain: Supervisor controls Vat and Operator controls BottlingLine.

Most of the semantic properties discussed above are not directly supported by current object-oriented programming languages; however, as they impose

important constraints on the implementation, they should be explicitly captured in an object-oriented model.

6.2 Design rationale for composite objects

Just as the properties of each whole-part structure must be documented to guide the implementation process, so the purpose of each composite object must be documented to help understand the design model and its derivation from the analysis model. To illustrate the approach, the rationale for each whole-part structure in the BottleFillingSystem is discussed, and the semantic properties of each are documented.

We proceed top-down, depth-first down the object composition hierarchy in Figure 7.

Whole:	*Bottle Filling System (models entire system)*
Parts:	*Vat, Supervisor, OperatedLines*
Kind:	*Assembly*
Rationale:	*Top-down Decomposition: System Partitioning into separate, cohesive parts*
Properties:	*Mutual inseparability, Configuration, Behaviour delegation*

The whole system is modelled as an assembly, whose components are subsystems with separate functional responsibilities. This structure is derived from the analysis model in three steps:

1. Partition the classes into a small number of groups, so as to minimize the number and strength of the inter-group links (Figure 11). WPAs bind more strongly than other associations - this is why filling Valves end up in the same partition as the BottlingLines. If shared WPAs are involved, then assemblies are considered stronger than groups and groups stronger than tuples. These guidelines help to minimise interactions amongst different branches of the hierarchy.

2. Select a key class in each group and model the whole system as a composite formed by objects of these classes (Figure 12). BottlingLine has been renamed OperatedLine to better convey its role.

3. Introduce a new object to manage the collection of OperatedLine objects (Figure 7).

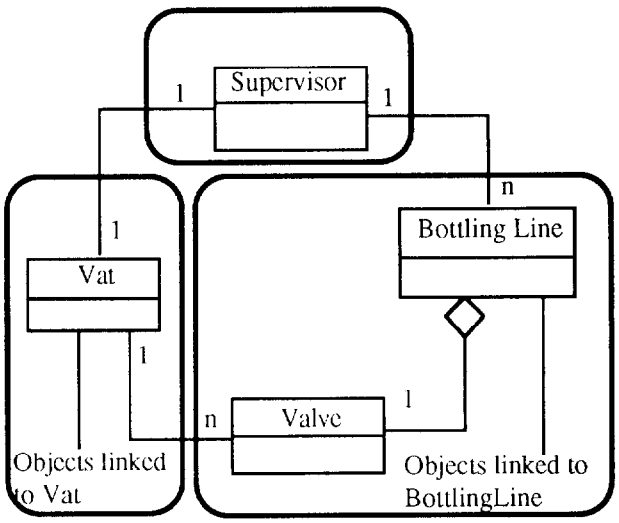

Figure 11: Partitioning the model to identify top-level abstractions

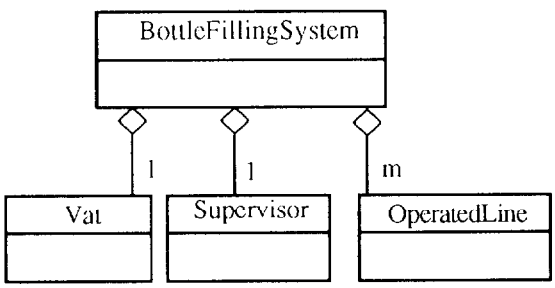

Figure 12: The system modelled as a composite object

Whole:	*OperatedLines*
Parts:	*OperatedLine*
Kind:	*Group*
Rationale:	*Simplify top-level decomposition*
Properties:	*Ownership, Constraint maintenance, Message broadcasting, Configuration*

This composite object does not model a specific entity in the problem domain. It is used to collect together all the BottlingLines (and their Operators), in order to simplify the top-level system structure by taking charge of the management of the OperatedLine objects. It ensures that all its member objects have the same value for their status attribute

(disabled/enabled/suspended). To maintain this constraint, OperatedLines is responsible for *broadcasting* supervisor messages to its members. OperatedLines does not encapsulate its members, so the Supervisor can share a reference to a single OperatedLine if necessary. Also no internal links between OperatedLine objects are required, as the lines in the plant operate independently from each other. However, each BottlingLine within each OperatedLine needs a link to the Vat to find out the liquid's pH to print on the label. Such a link is established as follows: BottleFillingSystem passes a reference to the Vat to its OperatedLines component, which in turn broadcasts it to each OperatedLine, and so on. So the composition hierarchy is used recursively to configure objects that need links to others in different branches of the hierarchy.

As a result of this transformation, BottleFillingSystem is no longer a combination of an assembly and a group, but just an assembly: the grouping responsibility having been delegated down to the new object. This reflects more accurately the meaning of the top-level decomposition: even though individual lines are dispensable and not functional components of the system as a whole, the set of lines is a functional component of the system. Objects modelling entire systems can often conveniently be modelled as assemblies of functional components. Any grouping composites can be pushed one level down in the part-of hierarchy by introducing new abstractions.

Whole:	*OperatedLine*
Parts:	*BottlingLine, Operator*
Kind:	*Assembly*
Rationale:	*Encapsulation of association*
Properties:	*Mutual inseparability, Configuration, Propagation*

This aggregate does not correspond to an entity in the problem domain. Its main purpose is to *encapsulate* its two part objects and their association, to decouple them from the OperatedLines collection. It propagates messages coming from the OperatedLines collection to the Operator object, which communicates them to the human operator and starts/stops the BottlingLine as appropriate. An OperatedLine object conceals and manages a BottlingLine-Operator link.

Whole:	*BottlingLine*
Parts:	*Labeller, Filling Station, Gate, Button removeSignal*
Kind:	*Assembly*
Rationale:	*Assembly from problem-domain model*
Properties:	*Encapsulation, External configuration of all parts; Mutual inseparability; Configuration, Behaviour delegation and control of FillingStation*

BottlingLine is derived from the analysis model. The three devices directly involved in the filling process have been grouped into a new assembly, the Filling Station.

BottlingLine manages its components. In particular, BottlingLine is the only object that can generate events (e.g., stop_filling, start_filling) for the FillingStation. All the part objects (except the FillingStation, see below) are device interface objects, i.e., they interface to a concrete device. In this system, all such objects are passive and have no knowledge of their function within the problem domain, whereas functional aggregates are often active and embody crucial domain knowledge (e.g., the BottlingLine knows that when a bottle is removed the gate should be opened). This approach enhances the reusability of the interface objects and decreases design complexity by limiting the number of objects with control responsibilities.

As another example of the allocation of problem-domain knowledge, the Labeller does not know what values it is printing on the labels nor where they come from. This knowledge pertains to the BottlingLine, which has a link to the Vat to find out the values to be printed.

Whole:	*Filling Station*
Parts:	*OnOff Sensor contactSensor, AnalogueSensor Scales, Valve fillingValve*
Kind:	*Assembly*

Rationale: *Algorithmic decomposition: simplifies dynamic behaviour of BottlingLine*
Properties: *Encapsulation, Mutual inseparabilty, External configuration of all parts*

The FillingStation is a conceptual abstraction with no corresponding tangible entity in the problem domain. It manages the passive objects interfacing to the real devices directly involved in filling bottles with liquid. The FillingStation exhibits *behavioural* cohesion. In other words, there exists a process in the system - "fill one bottle" - that calls at frequent intervals the services of its three parts. This process is encapsulated by the Filling Station object. Its existence simplifies, by decomposition, the dynamic behaviour of the BottlingLine object. This delegates responsibility for bottle filling to the active FillingStation, while retaining responsibility for starting and stopping the filling process and interacting with the operator and the other devices in the bottle filling line.

Whole: *Vat*
Parts: *PHControl, LiquidLevelControl*
Rationale: *Models a tangible object in the problem domain and its attributes*
Breaks the vat control into two concurrent activities
Kind: *Assembly*
Properties: *Encapsulation*
(Supervisor visible to PHControl)
Ownership,
Dynamic behaviour delegation

The Vat is a problem-domain object. Its parts are derived from what were attributes in the analysis model: the liquid pH and the liquid level. The reason for promoting these to the rank of part objects is that each is associated with a separate system activity. Furthermore the two activities can be described and implemented as concurrent processes.

Whole: *PHControl (LiquidLevelControl)*
Parts: *PHSensor, PHValve (LevelSensor, InputValve)*
Kind: *Assembly*
Rationale: *Models a property of the Vat associated with a control process*
Separates essential function from implementation mechanism
Properties: *Encapsulation,*
Mutual Inseparability,
External configuration

These two composite objects do not model tangible objects but concurrent system functions. Their parts model the devices used in each control function. Encapsulating the devices within each control object separates essential system functions from their implementation, an approach consistent with the separation of essential and implementation modelling of Real-Time Structured Analysis and Design [Ward and Mellor 85]. Objects clearly related to system goals are more stable than objects modelling physical devices that are part of the solution space. For example, if it was required to measure the pH via multiple sensors to increase accuracy, the change would be limited to the implementation of pHControl and would not affect its external interface to the Vat.

7 Conclusions

Current object-oriented methods and languages are not expressive enough to represent the richness in semantic properties and development roles of composite objects.

We have argued that treating composite objects separately from problem-domain class associations and explicitly capturing their design role, as well as their problem-domain semantics, helps to separate analysis and design concerns and to document the rationale for important modelling and design decisions that might otherwise be left unrecorded.

We have illustrated how whole-part associations can model different types of problem-domain relationships, and how object composition can be used to evenly distribute structural, functional and control complexity in a model.

A useful spin-off of capturing the design properties of composite objects explicitly in a model is to enable checking of a model for semantic consistency between

the behavioural and structural view. For example, scenarios of object interactions can be checked for consistency with the stated structural properties (whether the visibility properties are complied with, whether creation and deletion of objects is compatible with the separability, ownership and immutability properties, etc.). CASE tools for OOD should automate as much of this as possible.

The example used in this paper has illustrated the analysis and design roles of composite objects particularly applicable to the domain of embedded monitoring and control systems with a fairly static configuration. We believe more research is required to analyse and streamline the use of composite objects in more dynamic environments, where objects and links are frequently created and deleted at run-time. The concepts of encapsulation and separability, in particular, must be refined to account for their temporal dimension. It must be possible, for example, to model the migration of objects from one composite to another.

We also believe that many of the properties that we have classed as design properties, such as separability and immutability, can apply to real-world entities as well as software objects. Thus they can be investigated before software concerns are addressed. However, since software objects often do not exhibit the same properties as their real-world counterparts, we believe that a better than currently available understanding of the model transformation process that takes place during design is required, in order to account for differences between the analysis and design model. The method presented here does not address such issues, although it provides a framework in which they can be explored.

Acknowledgements

I am grateful to Richard Mitchell for his collaboration in the original design of the Bottle Filling System and for his tireless support and useful comments.

References

Atkinson C. 1991. *Object-Oriented Reuse Concurrency and Distribution. An Ada-based approach.* ACM Press. Addison-Wesley.

Booch G. 1991. *Object oriented design with applications.* Benjamin Cummings.

Coad P. and Yourdon E. 1990. *Object-oriented analysis, 1st ed.*, Yourdon Press/Prentice-Hall.

Coad P. and Yourdon E. 1991. *Object-oriented analysis, 2nd ed.*, Yourdon Press/Prentice-Hall.

Coleman D., Hayes F. and Bear S. 1992. Introducing ObjectCharts or how to use Statecharts in Object-Oriented Design. *IEEE Transactions in Software Engineering*, **18**(1), 9-18

de Champeaux D. 1991. Object-Oriented Analysis and Top-Down Software Development. *Proceedings of the 1991 European Conference on Object-Oriented Programming*, Springer-Verlag, 360-376.

de Champeaux D., Lea D. and Faure P. 1992. The Process of Object-Oriented Design. *Proceedings of OOPSLA '92*, ACM, 45-61.

Harel D. 1987. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming*, **8**(3), 231-274.

Hartmann T., Jungclaus R. and Saake G. 1992. Aggregation in a Behaviour Oriented Object Model, *Proceedings of the 1992 European Conference on Object-Oriented Programming*, Springer-Verlag, 57-77.

Hogg J. 1991. Islands: Aliasing Protection In Object-Oriented Languages, *Proceedings of OOPSLA '91*, ACM, pp. 271-285.

Hull R. and King R. 1987. Semantic Database Modelling: Survey, Applications and Research Issues. *ACM Computing Surveys*, **19**(3), September 1987.

Jacobson I. 1992. *Object-Oriented Software Engineering. A Use Case Driven Approach.* Addison-Wesley.

Jungclaus R., Saake G. 1991. Formal Specification of Object Systems, in *TAPSOFT '91, Proceedings of the International Joint Conference on Theory and Practice of Software Development*, Goos G. & Hartmanis J. eds., Springer-Verlag, 60-82.

Kramer J., Magee J., Sloman M. and Dulay N. 1992. Configuring object-based distributed programs in REX. *Software Engineering Journal* March 1992, 139-149.

Monarchi D.E. and Puhr G.I. 1992. A Research Typology for Object-Oriented Analysis and Design. *Communications of the ACM*, September 1992, **35**(9), 35-47.

Odell J. 1992. Managing object complexity, part II: composition. *Journal of Object-Oriented Programming*,, **5**(6), October 1992, 17-20.

Robinson P. ed. 1992. *Object-oriented Design*. Chapman & Hall.

Rubin K. and Goldberg A., 1992. Object Behaviour Analysis. *Communications of the ACM*, September 1992, **35**(9), 48-62.

Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W. 1991. *Object-oriented Modeling and Design*. Prentice-Hall.

Ward P.T. and Mellor S.J. 1985. *Structured Development for Real-Time Systems. Vol. 1-3*. Yourdon Press.

Winston M.E., Chaffin R. and Herrmann D. 1987. A Taxonomy of Part-Whole Relations. *Cognitive Science*, **11**, 417-444.

Wirfs-Brock R., Wilkerson L. and Wiener L.1990. *Designing Object Oriented Software*.Prentice-Hall.