JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 10, November-December 2004

UML 2 Composition Model

Conrad Bock, U.S. National Institute of Standards and Technology

The composition model in the Unified Modeling Language, version 2 (UML 2), is a major upgrade to the familiar "black diamond" composition of earlier versions. It supports connections between parts at the same level of decomposition, in addition to the usual part-whole associations. Complex networks of entities can be represented within a single class, inherited to subclasses, with links maintained between objects playing parts at runtime. The model also supports connections between parts of parts (ports), enabling more detailed structural modeling and message forwarding, which increases independence of reused applications and provides better plug-compatibility for components.

1 INTRODUCTION

UML 2 composite structures add new concepts and a new diagram to conventional class modeling [1]. To justify this, it is important to show existing techniques are not adequate. Section 2 demonstrates some major difficulties in using the UML 1.x composition model on a simple example. It is possible with significant effort to achieve the same effect as UML 2 composition with UML 1.x constructs, but the extent of the effort justifies the new features. Showing a mapping from new to existing constructs also ensures the new ones are well understood.

Section 3 introduces the most basic aspects of UML 2 composition, and shows it is much simpler and more powerful than UML 1.x. Section 4 describes some computational services supported by the new model and interprets some of these as constraint maintenance. Section 5 covers the model for parts of parts (ports), and Section 6 shows how they are used to provide better encapsulation of components with message forwarding. Section 7 covers inheritance of composition models. Section 8 summarizes the article[1].

[1] The exposition happens to start with purely structural aspects, using physical examples, and proceeds to communication, with software examples, in the usual fashion of building behavior on structure. However, UML 2 composition supports communication with more independence from structure than UML 1.x (see Section 6), so other expositions might take the opposite path. The article also begins with class modeling, since it is probably most familiar to readers, and shows how role modeling extends it. Other explanations might begin with role modeling and show how class modeling is created from that.

Cite this column as follows: Conrad Bock: "UML 2 Composition Model", in *Journal of Object Technology*, vol. 3, no. 10, November-December 2004, pp. 47-73.
http://www.jot.fm/issues/issue_2004_11/column5

2 CLASSES AND ASSOCIATIONS NOT ENOUGH

UML 1.x composition is familiar to users as the black diamond notation on associations (called *composite aggregation*, or informally "strong composition") [2], for example as shown in Figure 1. The figure says that boats and cars have engines, that cars have wheels, boats have propellers, and engines are similarly decomposed. The black diamonds mean the same instance of ENGINE cannot be used simultaneously in an instance of CAR and an instance of BOAT, even though the class ENGINE is shared between the classes CAR and BOAT. This is why the multiplicities from ENGINE to BOAT and CAR are optional. The black diamond also means that an instance of ENGINE is destroyed when its containing instance of CAR or BOAT is destroyed. UML 2 is backwardly compatible with this model[2].

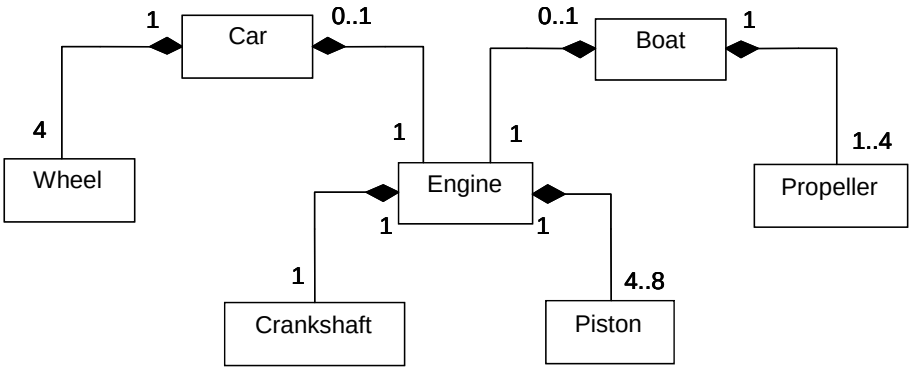


Figure 1: UML 1.x Composition

While the UML 1.x model is fine for hierarchical decomposition, as in Figure 1, it has significant limitations when connecting elements at the same level of decomposition, for example, as attempted in Figure 2. The additional associations are trying to say that engines power wheels in cars, and engines power propellers in boats. However, the associations are defined globally for all engines and boats, not in the context of individual cars and boats. This means:

1. The engine in one car can power the wheels in another, instead of just the wheels in the car that has the engine. The composition associations from CAR and BOAT do not limit the POWERS association (which is not legal UML anyway, see discussion of Figure 4 in this section).

[2] Classes in the physical examples of this article only describe information records of real or imagined physical objects. In particular, destruction in UML refers to the deletion of information instances, not necessarily physical ones [3]. UML does not specify whether destroying the information instance causes the destruction of the physical object, or whether the UML semantics for destruction applies to physical objects. For example, an insurance company may view a "totaled" car as destroyed, even though the engine is not affected. UML does not address these and other issues of composition [4].

2. Each engine is required to power both propellers and wheels, without regard to whether the engine is in a car or a boat. The multiplicity is mandatory from ENGINE to PROPELLER and WHEEL.[3]

3. An engine can power the two right wheels of the car, instead of the front wheels. The POWERS association does not indicate which two of the four wheels should receive power.

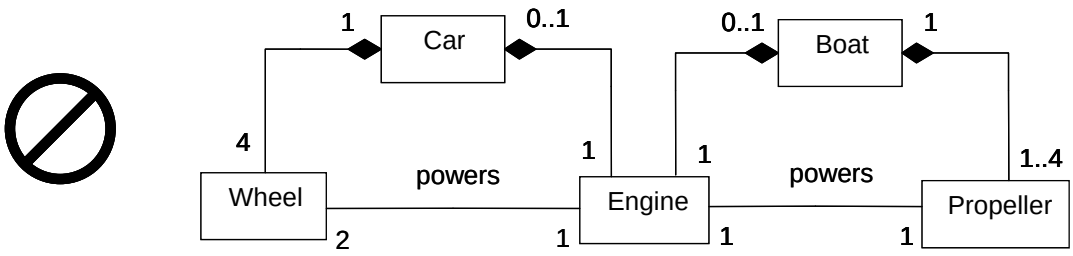


Figure 2: Antiexample for Associations at the Same Level of Decomposition

The problems above are shown by the instance model in Figure 3, which is allowed under the class model of Figure 2. The names of instances are given to the left of the colons, and the classes they are instances of to the right. Links between instances are notated the same way associations are, except multiplicity does not apply, because links always have one instance on each end. The E1 instance of ENGINE is contained by MYCAR, but powers wheels contained by YOURCAR. The engine E2 is powering the correct propeller P1, but also powers wheels RIGHTFRONT1 and RIGHTBACK1. Finally, the right two wheels of YOURCAR receive power instead of the front two.

[3] The multiplicities of POWERS from ENGINE could be made optional, but then cars and boats could have engines that do not power anything.

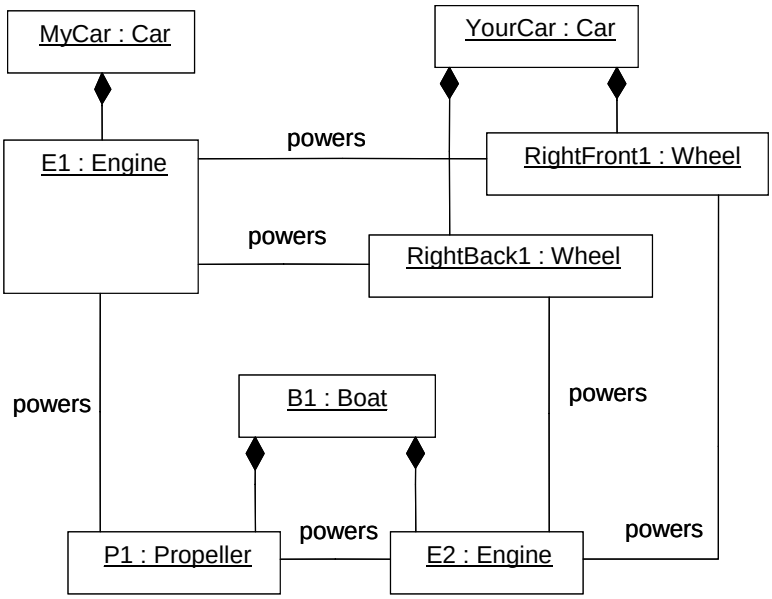


Figure 3: Instance Model Allowed under Figure 2

We might address these issues with generalized classes and associations as shown in Figure 4. The generalized classes POWERSOURCE, POWERTRANSMITTER, are marked as abstract, which means instances can only be created from their subtypes, not from themselves directly. However, the instances of the subtypes are still instances of POWERSOURCE and POWERTRANSMITTER indirectly. For example, boat engines are still power sources, even though POWERSOURCE cannot have direct instances[4]. A generalized POWERS association is introduced between the generalized classes POWERSOURCE and POWERTRANSMITTER. It is abstract and its links (instances) are derived as the union of links of the specialized associations in the subtypes. Instances cannot be linked with the abstract POWERS association, only by the specialized associations. Moving the mandatory multiplicities in Figure 2 to a more general POWERS association resolves the problem that Figure 2 requires engines to power both wheels and propellers (which is not legal UML anyway, because binary associations relate exactly two classes[5]).

To ensure that car engines only power car wheels, and boat engines only power propellers, the abstract classes and associations in Figure 4 must be specialized [5], using the association specialization capabilities introduced in UML 2. The REDEFINES property on association ends indicates that the association is restricted to the class at that end, and renamed for that purpose. For example, the redefinitions on SOURCE and TRANSMITTER between CARENGINE and FRONTWHEEL mean that car engines can only power front car

[4] If wheels were used in other classes besides cars, or propellers used in other classes besides boats, they would need to be specialized also. Specialized cars would also be needed to model rear-wheel drive cars.

[5] UML supports multiple associations of the same name, but in Figure 2 we assume the modeler is attempting to use the same association twice.

wheels, which can only be powered by car engines[6]. Similar redefinitions are used to ensure that only boat engines power propellers, and vice versa. Multiplicities not specified are inherited from the more general association, for example, for the INCAR ends. The composition associations must also be specialized, as shown in Figure 5. These limit the power sources of cars and boats to car engines and boat engines respectively, and limit the transmitters to front wheels and propellers respectively.[7]

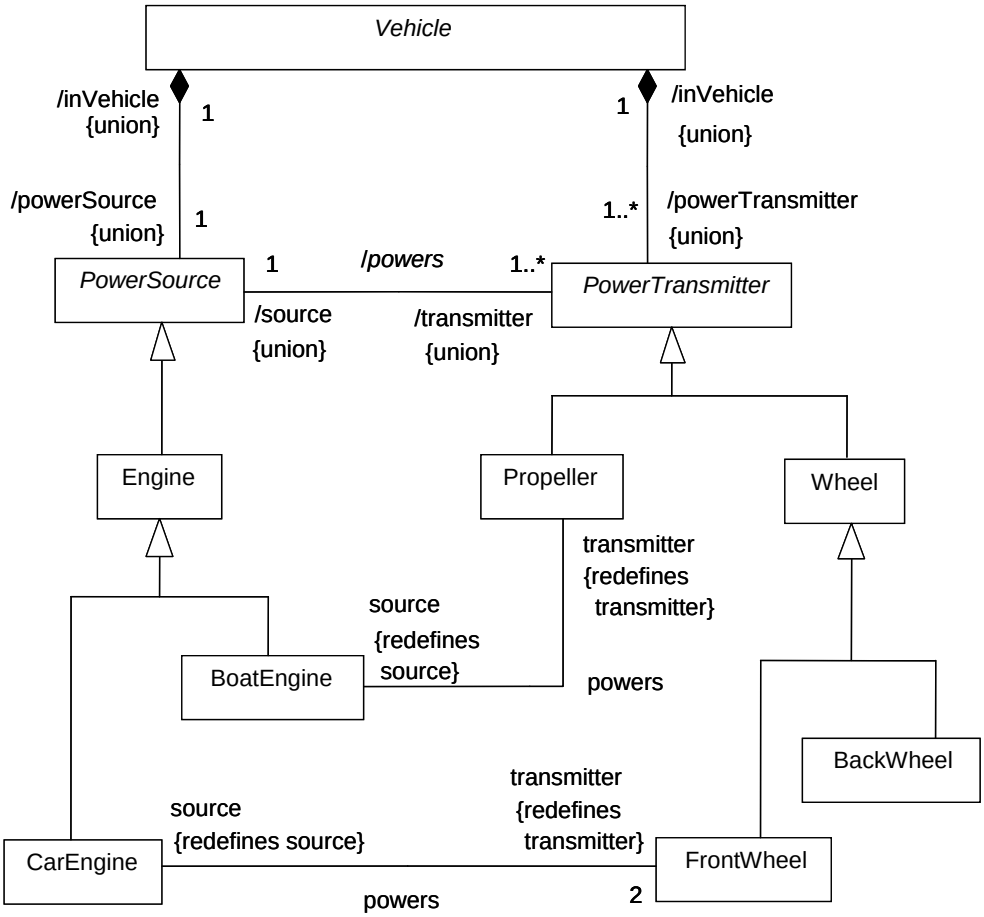


Figure 4: Specialized Classes and Associations at Same Level of Decomposition

[6] However, links of the abstract POWERS association can still be read. This preserves substitutability. For example, suppose a program generated from this model referred to a variable with type POWERSOURCE, understanding that it actually was an instance of CARENGINE or a BOATENGINE at runtime. The program could read the TRANSMITTER association on the variable, because this is the same on all the subtypes, but cannot change it, because the specific subtype is not known at programming time.

[7] The composition associations of Figure 5 can be generalized to a single compositional association, which is useful in sketching composition hierarchies, sometimes informally called the "has a" relation. This technique is used in the UML 2 metamodel at the top of the class hierarchy, where the metaclass ELEMENT owns itself. It supports the containment hierarchy needed for interchange file formats.

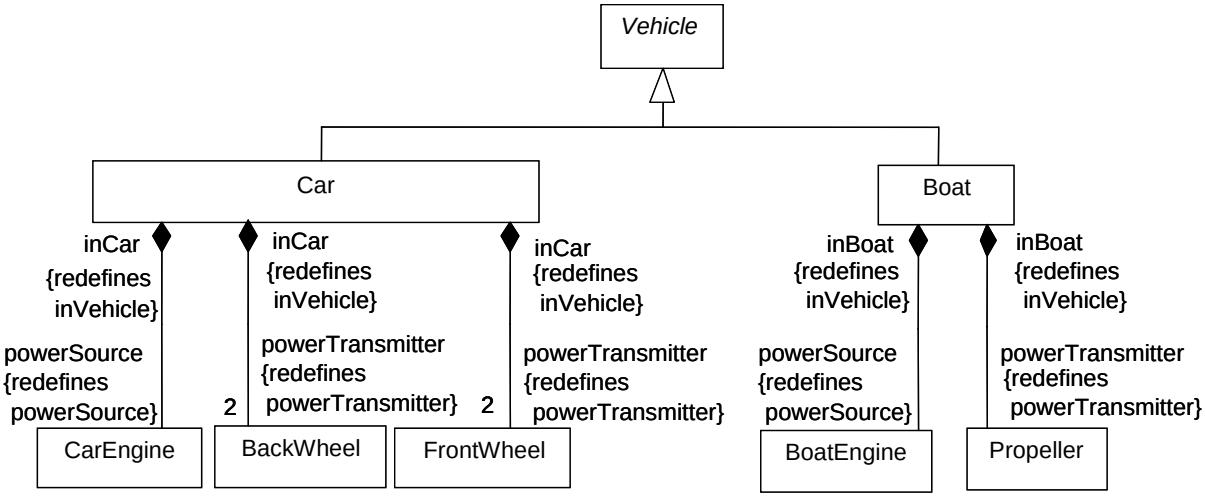


Figure 5: Specialization of Composition Associations from Figure 4

However, even the cumbersome class models of Figure 4 and Figure 5 do not prevent an engine in one car from powering wheels in another. This requires each individual car to have classes for its particular wheels and engine, so the association can be further specialized in each car [6]. Although this is theoretically correct, it is obviously impractical, showing that classes and associations cannot model composite structure efficiently.

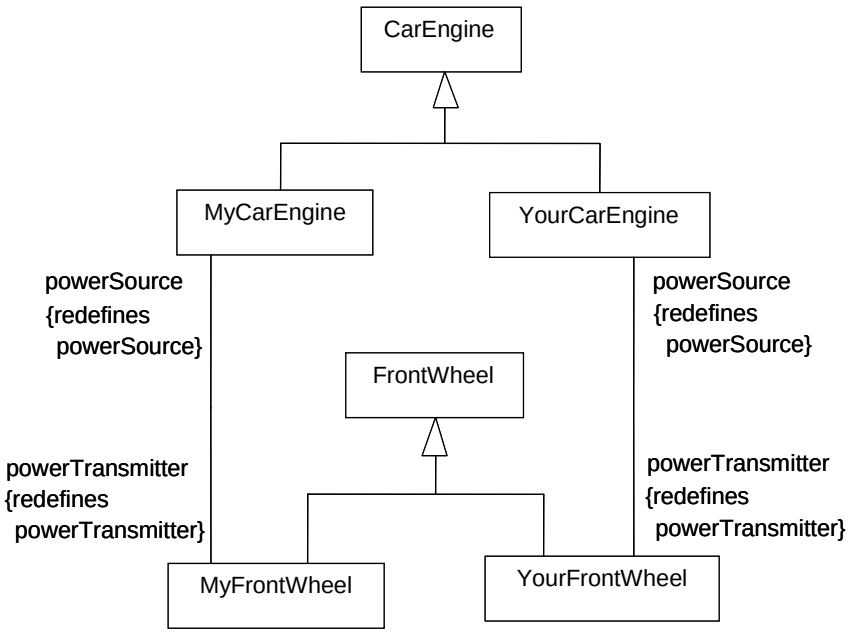


Figure 6: Specialized Classes and Associations for Each Instance

3 ASSOCIATIONS IN CONTEXT

UML 2 addresses the problems of using class diagrams for composition by introducing a new diagram specifically for composite structure, as shown in Figure 7. The basic principle is to define usages of classes and associations in a context, rather than associating classes globally. The contexts are also classes, like CAR and BOAT, while the usages of other classes are shown as rectangles inside the containing classes, and the usages of associations are shown as line segments. The rectangles are labeled with a colon notation to distinguish classes being used, such as WHEEL, to the right of the colon, from how they are used, such as FRONT, to the left of the colon. The rectangles are distinguished from instances by not having underlined labels. The colon notation is used on the line segments also, where the association being used is shown to the right of the colon, and how it is used to the left, which may be anonymous. The numbers inside the rectangles specify the number of instances that will be used, for example, two front wheels in a car.

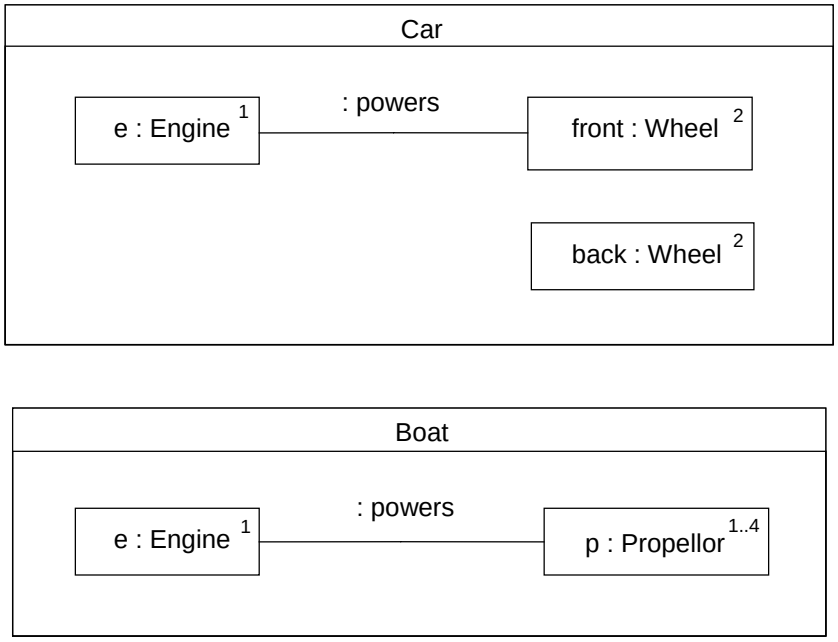


Figure 7: UML 2 Composition

Figure 7 means[8]:

In each car:

1. There will be one engine, two front wheels, and two back wheels.
2. The engine will power the front wheels in the same car as the engine.
3. The engine will not power anything else in the car, other cars, or boats.

[8] Figure 7 also establishes rules on communication between entities, which are constrained to follow connectors. See Section 6.

In each boat:

1. There will be one engine and one or more propellers.
2. The engine will power the propellers in the same boat as the engine.
3. The engine will not power anything else in the boat, other boats, or cars.

To enforce the above rules, the model must provide a way to identify the objects contained by individual cars and boats, so the engine in one car can be distinguished from the engine in another, and front wheels distinguished from back wheels, and so on. This is done through association ends, which can represent navigation (mappings) from individual cars and boats to the particular objects they contain. For example, Figure 8 elaborates Figure 1 and the top portions of Figure 4 and Figure 5, to define the association ends E, P, FRONT, and BACK. The SUBSETS property on association ends is another association specialization capability in UML 2. It indicates the association is specialized and renamed at that end, but does not restrict the inherited association to the class at that end (compare to REDEFINES). For example, the INCAR end from ENGINE to CAR is subsetted from INVEHICLE, which means other vehicles besides cars can use engines, such as boats. The INCAR end can be navigated to find the car that a particular engine is in, if any, or INVEHICLE can be navigated to find whatever vehicle an engine is in, regardless of the type of vehicle[9].

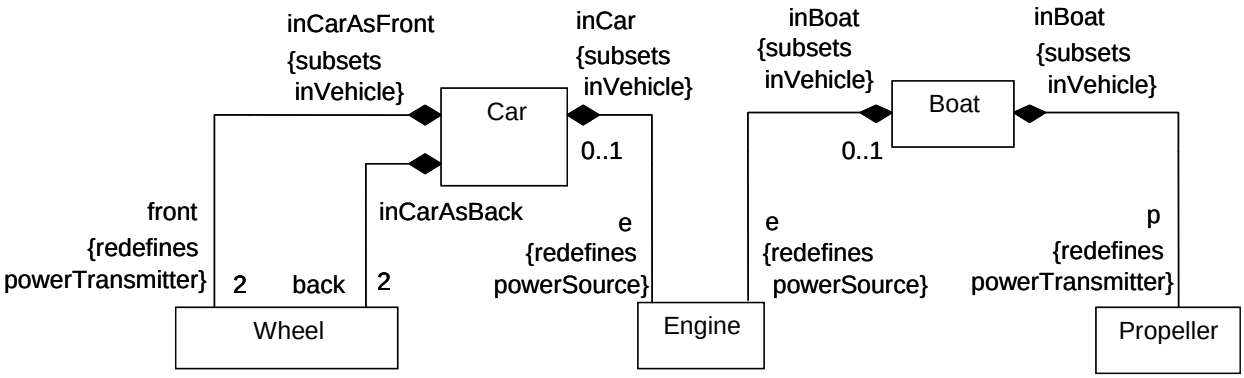


Figure 8: Composition Associations for Figure 7

The embedded rectangles in Figure 7 are another notation for the association ends opposite CAR and BOAT in Figure 8 (or properties in general, see Section 4 and the example in Figure 17). This is why the ends have the same names and multiplicities as the usages in Figure 7. Names of the usages are shown just to the left of the colon, namely, E, FRONT, and BACK. Embedded rectangles do not represent classes, as rectangles do in class diagrams, to avoid the problems of using classes for composition cited in Section 2.

[9] The INCARASBACK end could be specialized from INVEHICLE if INVEHICLE was generalized to all parts of CAR. See footnote 7.

Figure 9 shows example instances of CAR and BOAT, with links from an individual car MYCAR to its individual engine, front wheels, and back wheels, and similarly for YOURBOAT. Figure 10 is an alternate notation for Figure 9, where the instance name is given to the left of the slash, the association end name to the left of the colon, and the class being reused to the right of the colon. Navigation occurs by starting with MYCAR, traversing associations along their ends, and arriving at the objects that are contained by MYCAR in specific ways. The results of navigation, such as E1:ENGINE, W1:WHEEL, and W2:WHEEL are related by POWERS links according to the composite structure defined in Figure 7 (compare to Figure 3). Notice that two association ends can have the same name in different composite structures, but are treated separately. In this example, two association ends named E are used by CAR and BOAT, but are distinguished by the context.

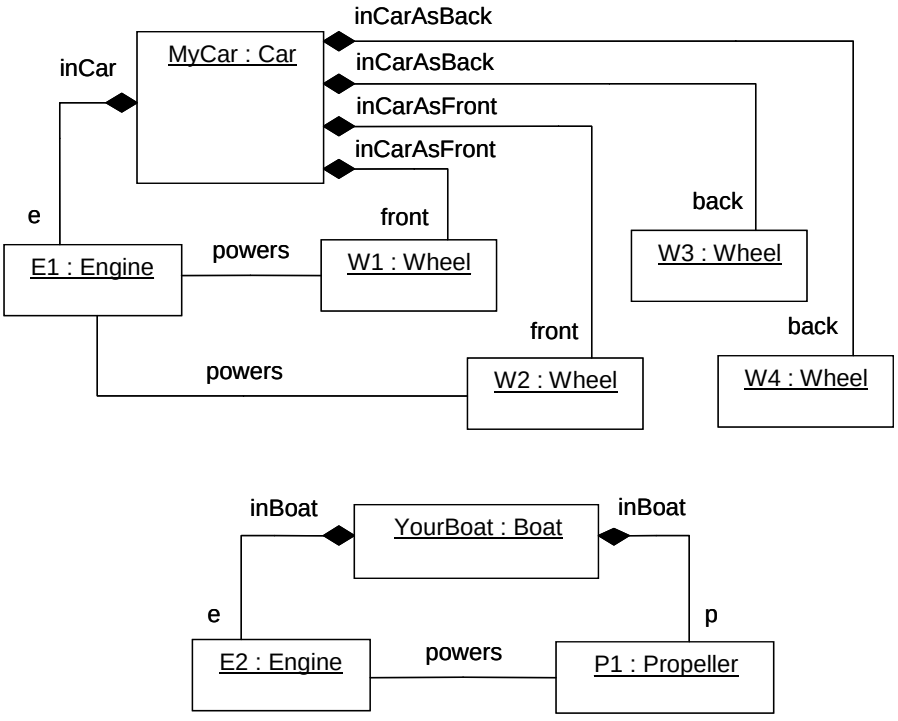


Figure 9: Instance Model Allowed under Figure 7 and Figure 8

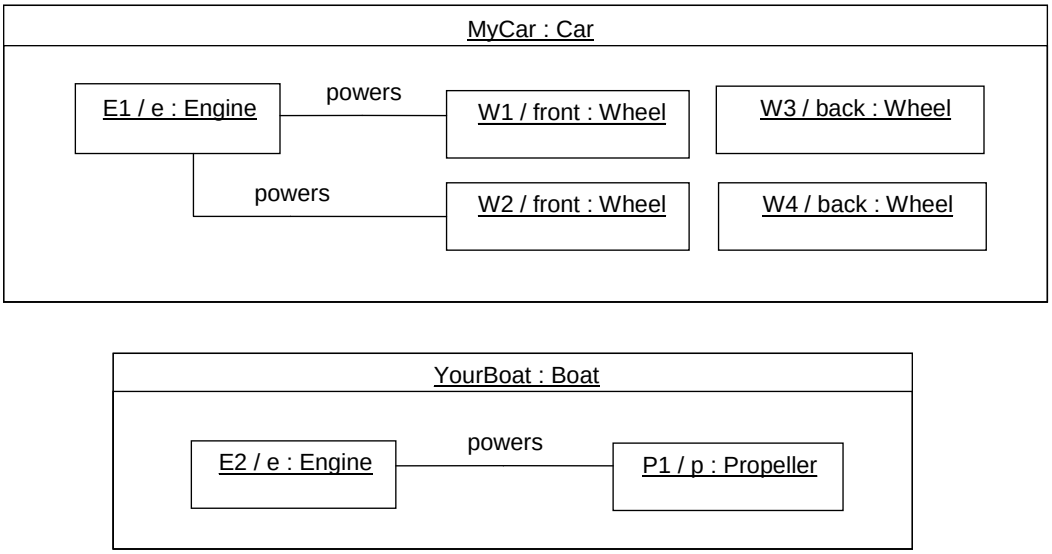


Figure 10: Another Notation for Figure 9

In UML 2 the composite (strong composition) association ends opposite context classes are called *parts*, as in part in a play, rather than part as an object[10]. For example, in Figure 8 the ends E, FRONT, and BACK are parts. This term applies to composite association ends in a class diagram as well as embedded rectangles in composite diagrams. Informally, an object playing a part is sometimes called a "part", but this makes it difficult to understand that composite structures connect objects based on how they participate in the context object, not based on the individual objects themselves. For example, the conversation in a play is determined by the parts performed by the actors, not by the actors themselves, which change over the many performances of the play[11].

The line segments in Figure 7 are notation for a new concept in UML 2 called *connector*. Connectors are usages of associations, and relate parts (composite association ends) of a context class. They relate parts, rather than classes, to navigate to objects contained by instances of the context class, and link the resulting objects together. For example, in Figure 7, the connector between the E and FRONT parts of CAR means that instances of the POWERS association (links) are established between the objects playing those parts in each individual car. Connectors can also relate non-composite associations and attributes, as discussed below.

The use of association ends and connectors can be understood as shorthand for the very specialized classes and associations in Figure 6 [6]. Instances of the specialized classes, such as MYCARENGINE and MYFRONTWHEEL, are the same as the result of

[10] The composition model also supports noncomposite associations and attributes as well. See Section 4.
[11] The term "role" might also be used informally, but it is equally overloaded. UML 1.1 called association ends "roles", but this was not always used in the above sense of navigating from one end object to another. UML 1.3 changed the term to *association end*.

navigating association ends from individual cars and boats to the participating objects, for example E1:ENGINE, W1:WHEEL, and W2:WHEEL in Figure 9. The specialized associations have mandatory multiplicities that establish links between the specialized classes, which are the same links established by connectors between the objects that result from navigating association ends.

However, the UML 2 composition model does not automatically define the specialized classes in Figure 4 or Figure 6, such as CARENGINE. The specialized classes of Figure 4 are useful in applications that add characteristics to classes when they are used in composing others [7]. For example, an employee is a kind of person who is in an employment relationship with a company. The employee class may introduce new associations, for example to benefits. These "role classes" provide some useful contextualization, for example to place additional constraints on contained objects, see discussion of Figure 12 below.

Connectors can have multiplicities different from the association ends (parts) they relate. For example, a more general model of cars that covers both front-wheel drive and rear-wheel drive is shown in Figure 11. It uses a new association end W that navigates to all the wheels in a car, as a generalization of FRONT and BACK in Figure 7. The new end has a multiplicity of 4, but the POWERS connector to it has a multiplicity of 2 on that end. This means for each car, two of the four wheels in the car will be powered. Unlike Figure 2, the model intentionally does not specify which two wheels are powered, with the expectation that CAR will be specialized for front and rear-wheel drive cars (see Section 7 about inheritance of composite structure). When connector multiplicities are not specified, they are the same as the parts being connected, as in Figure 7.

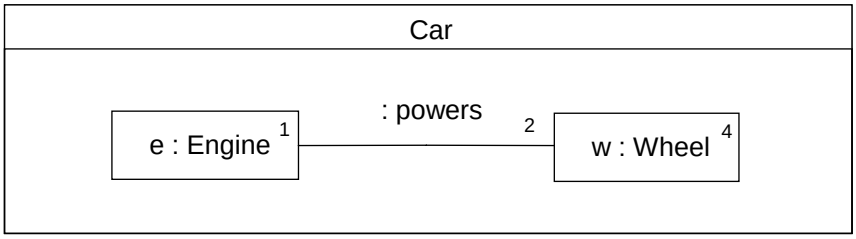


Figure 11: Connector End Multiplicity

Composite structure can also model software artifacts, for example, dialog boxes. When a dialog box comes up on a screen, it can be an instance of a class with composite structure. It contains controls such as text boxes, menus, and so on, which are instances of other classes, and that have specific attribute values for their position, labels, and so on. Tabbing relations link controls together. An example composite class for a dialog is shown in Figure 12. It uses a suggested presentation option that is not defined in UML 2, but would be intuitive for modelers, to show the initial values for the various control attributes. For example, the objects playing the NAME part are text boxes, and the initial attributes values for objects playing parts are listed in the attribute compartments, which can be omitted for compactness of display, of course. Connectors are usages of the

TABTO association defined between controls generally, not shown, and give the tabbing order for this particular dialog. The UML repository model for contextualized attribute values is not as simple as it could be, requiring additional parts for each value, and connectors from the dialog control parts to the value parts, or using specialized part types such as those in Figure 5 with default values for attributes. Hopefully this will be addressed in future revisions.

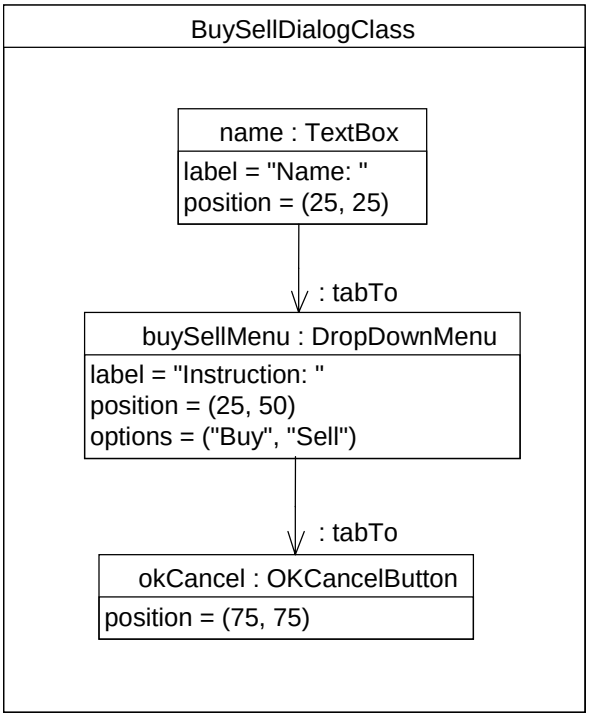


Figure 12: Composite Structure for a Dialog Box

A repository model for portions of Figure 7 and Figure 8 is shown in Figure 13. The context class CAR:CLASS owns the properties E:PROPERTY and FRONT:PROPERTY, which have values of type ENGINE:CLASS and WHEEL:CLASS, respectively. These properties are also the ends for the composition associations between CAR:CLASS and ENGINE:CLASS and WHEEL:CLASS, through the MEMBEREND associations. This unification of attributes and associations is new to UML 2[12]. The properties are related by an anonymous :CONNECTOR, which corresponds to the connector labeled `:powers` in Figure 7. Connectors have connector ends, to support multiplicities when they are different from the association ends being connected. The type of the connector is POWERS:ASSOCIATION, which will be instantiated to link the values of E:PROPERTY and FRONT:PROPERTY when they have values in an instance of CAR:CLASS.

[12] An attribute in UML 2 has the same semantics as a unidirectional association. The only difference in the repository model is whether a property is an end for an association or not.

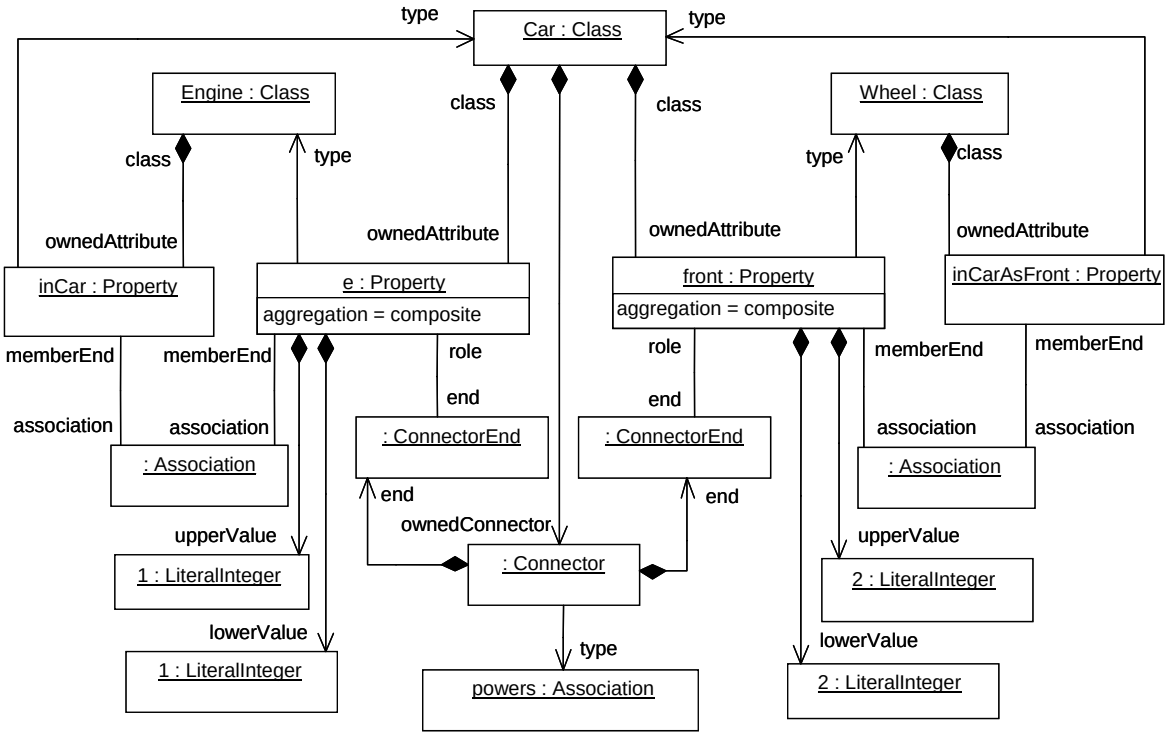


Figure 13: Repository Model for Portions of Figure 7 and Figure 8

4 COMPOSITION SERVICES

The UML 2 composition model enables useful runtime services, including:

1. Maintenance of links between objects playing parts
   When an object begins to play a part in an instance of a composite class, links can be established to the object according to the connectors of the composite structure. For example, based on Figure 7, when an engine is added to a car, it is linked to the wheels already in that car. Likewise, when the car is created, the engine and wheels used in the creation are linked.
   When an object stops playing a part in an instance of a composite class, links can be destroyed if they were established by the connectors of the composite structure. For example, when an engine is removed from a car, it is unlinked from the wheels of that car. Likewise, when the car is destroyed, the links between engine and wheels in the car are destroyed[13].

[13] UML 2 finalization has not yet provided an option on CreateLinkAction, AddStructuralFeatureAction, DestroyLinkAction, and RemoveStructuralFeatureAction for creating and destroying links based on connectors, but this can be done with CreateLinkAction and DestroyLinkAction.

2. Creation propagation
   When an instance of a composite class is created, objects playing parts with minimum multiplicity greater than zero can be created to satisfy the multiplicity. For example, when an instance of CAR is created, four wheels can be created, two playing FRONT part and two playing the BACK. Automatic creation is not required, because UML does not dictate when multiplicities are enforced. For example, a car can be created using existing wheels, or can be created without wheels, and new or existing wheels added later. Tools can provide their own options for this.

3. Destruction propagation
   When an instance of a composite class is destroyed, the objects playing strongly composed parts are destroyed. For example, destroying an instance of CAR will destroy the objects playing the E, FRONT, and BACK parts. Any objects that had been removed from the instance of CAR beforehand are not destroyed, including weakly composed or uncomposed objects, see below[14].

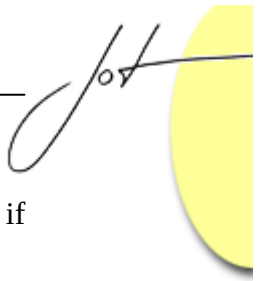
Contrary to the name, the composite structure model provides link maintenance with noncomposite associations and attributes. Connectors can relate:

- Shared aggregate ("weakly composed") association ends. The values of these association ends can be contained by more than one object at the same time. There is no destruction propagation required[15].
- Regular, noncomposed association ends. These do not require destruction propagation.
- Attributes with any kind of composition of their values, strong, weak, or none (composite, shared, or none).

The above are notated in a composite structure diagram with dashed line rectangles, as described later in Figure 17. Link maintenance applies to the these constructs, because it only depends on being able to identify objects related to the context object at runtime. This can be done by any kind of association end, as long as it is navigable from a context class to another, and any kind of attribute. For example, in Figure 17, establishing the link between computers uses the VISITOR and HOST association ends or attributes on instances of FTPSESSION (the composite structure diagram does not differentiate between attributes and association ends). Creation propagation can also apply to the above constructs, if the

[14] The finalization of UML 2 extended DestroyObjectAction with an option to destroy strongly composed objects using a single action. This is convenient for tools implementing strong composition (composite aggregation) semantics. Finalization has not yet provided an option on DestroyObjectAction for destroying links between non-strongly composed objects established by connectors, but this can be done with DestroyLinkAction.

[15] UML states that strong and weak composition are transitive, but this can only work for associations from a class to itself, and even then it would violate the single-owner rule for strong composition.



minimum multiplicity is greater than zero. Destruction propagation can apply to them if the minimum multiplicity on the context end is greater than zero, but is not required.

Link maintenance can be understood as an effect of enforcing constraints implied by a composite structure. For example, Figure 7 provides link maintenance equivalent to enforcing the constraint shown in

Expression 1, written in UML's Object Constraint Language (OCL) [8]. The `context` reserved word indicates the constraint holds for all instances of the class CAR, and the names used in the constraint refer to association ends and attributes of the same instance of CAR. The `inv` reserved word means the constraint holds over the lifetime of each instance. The constraint body follows, requiring a POWERS link between the value of the E property of each car and the values of the FRONT property. Equality means the constraint specifies all the POWERS links between the engine and front wheels, so they cannot participate in any other POWERS links. In particular, if there are no values in one of the properties on either side of the equality, there will be no powers link in the value of the other property. This reflects the third rule for cars implied by Figure 7[16].

```
context Car inv:
(e->notEmpty implies e.transmitter = front)
  and (front->notEmpty implies front.source = e)
```

Expression 1: Constraint Implied by Figure 7

Additional composition services are enabled by ports, see Section 6.

[16] This is a technique in OCL for a limited form of closed world, that is, if the constraint cannot establish a POWERS link between the engine and front wheels in a car, then the link does not exist. In pure first-order logic languages closure must be spelled out explicitly.

5 PORTS

Many applications require connections between parts of parts. For example, the simplified composite structure for a car in Figure 7 might be elaborated to model the parts of axles and wheels, as shown in Figure 14. The specification of the POWERS association in this example is changed to relate moving parts, as shown in Figure 15. Parts that are available for connection from outside of a composite are called *ports*.

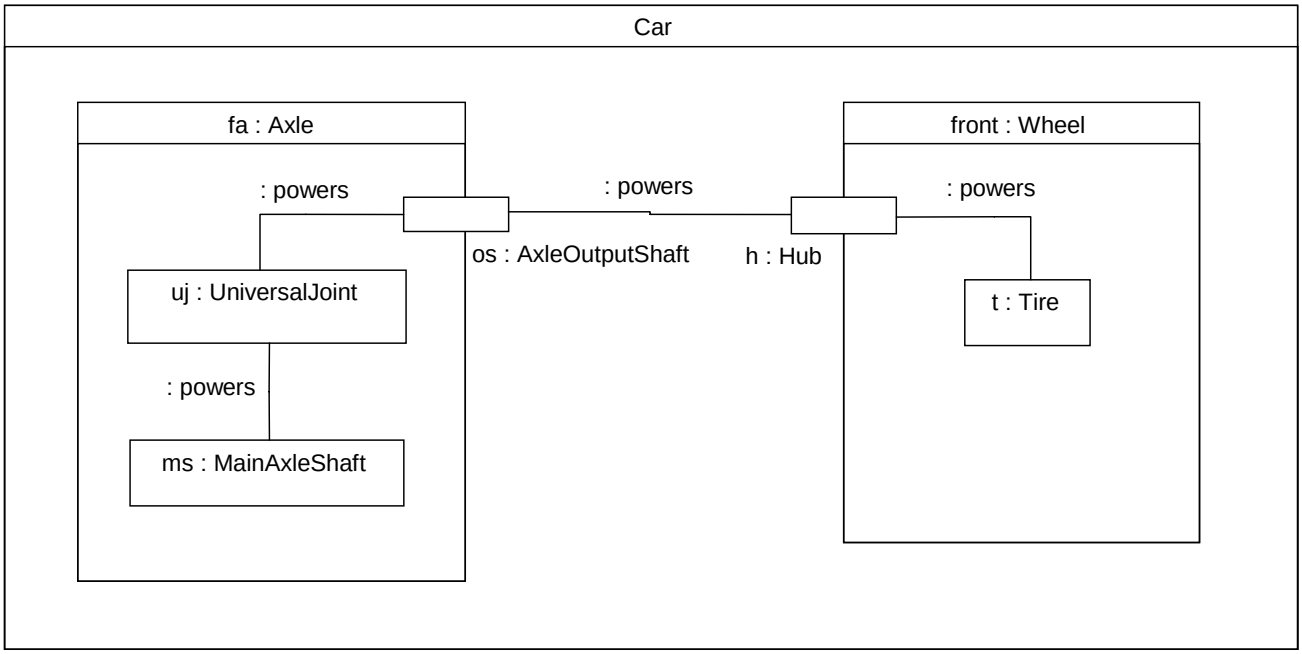


Figure 14: Ports

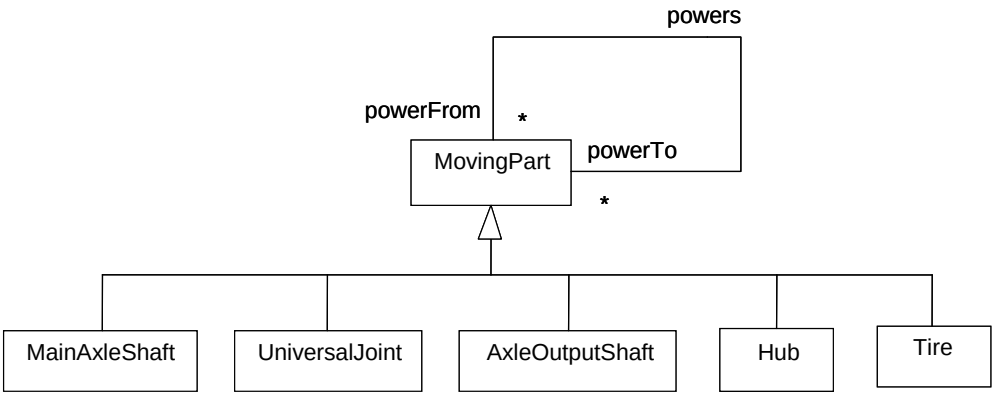
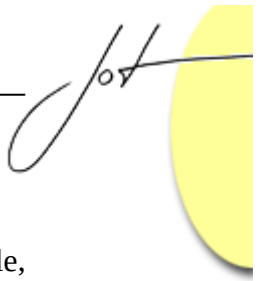


Figure 15: Classes and Associations Used in Figure 14



Connecting ports requires double navigation to find the objects to be linked. For example, to find instances to link according to the connector between OS:OUTPUTSHAFT and H:HUB requires navigation from an instance of CAR to the object playing the FA:AXLE part, and from that object to the object playing the OS:OUTPUTSHAFT part. The output shaft found this way is linked to the result of navigating from an instance of CAR to the object playing the FRONT:WHEEL part, and from that object to the object playing the H:HUB part[17].

Association ends require connectors to provide double navigation. For example, if Figure 14 did not have connectors, navigating across the association end FRONT:WHEEL identifies a wheel, but nothing prevents the second navigation of H:HUB from starting from some other wheel entirely. Double navigation requires that navigations of association ends are chained together, with the second end navigating from the results of the first. Connectors ensure that double navigation is used to establish links. This is important for applications that use the same type of part more than once in a composite structure. For example, there are multiple wheels on a car, so the single navigation provided by H: HUB is ambiguous about which wheel the hub is on.

Since ports are a special kind of part, the runtime effects of ports are inherited from parts: link maintenance, creation propagation, and destruction propagation. In particular, creation and destruction propagation recurs across multiple level of decomposition. For example, creating an instance of CAR optionally creates an instance of AXLE, which in turn creates instances of OUTPUTSHAFT, UNIVERSALJOINT, and MAINAXLESHAFT. Link maintenance applies to ports, for example when instances of OUTPUTSHAFT and HUB start and stop playing the parts OS and H, links between these are created and destroyed. Link maintenance between ports in this example is equivalent to enforcing the implicit constraint shown in Expression 2.

```
context Car inv:
fa->notEmpty and front->notEmpty
  implies
    (fa.os->notEmpty implies fa.os.powerTo = front.h
     and front.h->notEmpty implies front.h.powerFrom = fa.os)
```

Expression 2: Constraint Implied by Figure 14

The metamodel for ports is very similar to parts, but the metamodel for connectors is extended to specify which parts the connected ports are on (double navigation). Figure 16 shows a portion of the repository model for Figure 14. Most of it is the same as Figure 13, except that:

- ports such as OS:PROPERTY and H:PROPERTY are identified by the OWNEDPORT metaassociation end. This is a specialization of OWNEDATTRIBUTE.

[17] The double navigation requirement makes class models even more infeasible for composite structures, because a class would be needed for every part of part on every instance of the context class.

- connectors ends for ports have an additional metaassociation end PARTWITHPORT, which identifies the parts that the ports are on (double navigation).

The second aspect above means that a connector between ports of sibling parts actually refers to two navigations down from the owner of the parts, rather than one navigation from an instance of the port owner. For example, the connector in Figure 14 that has H:HUB on one end actually refers to a double navigation from an instance of CAR, through FRONT:WHEEL, not a single navigation from an instance of WHEEL. This is why the connector is owned by CAR:CLASS[18].

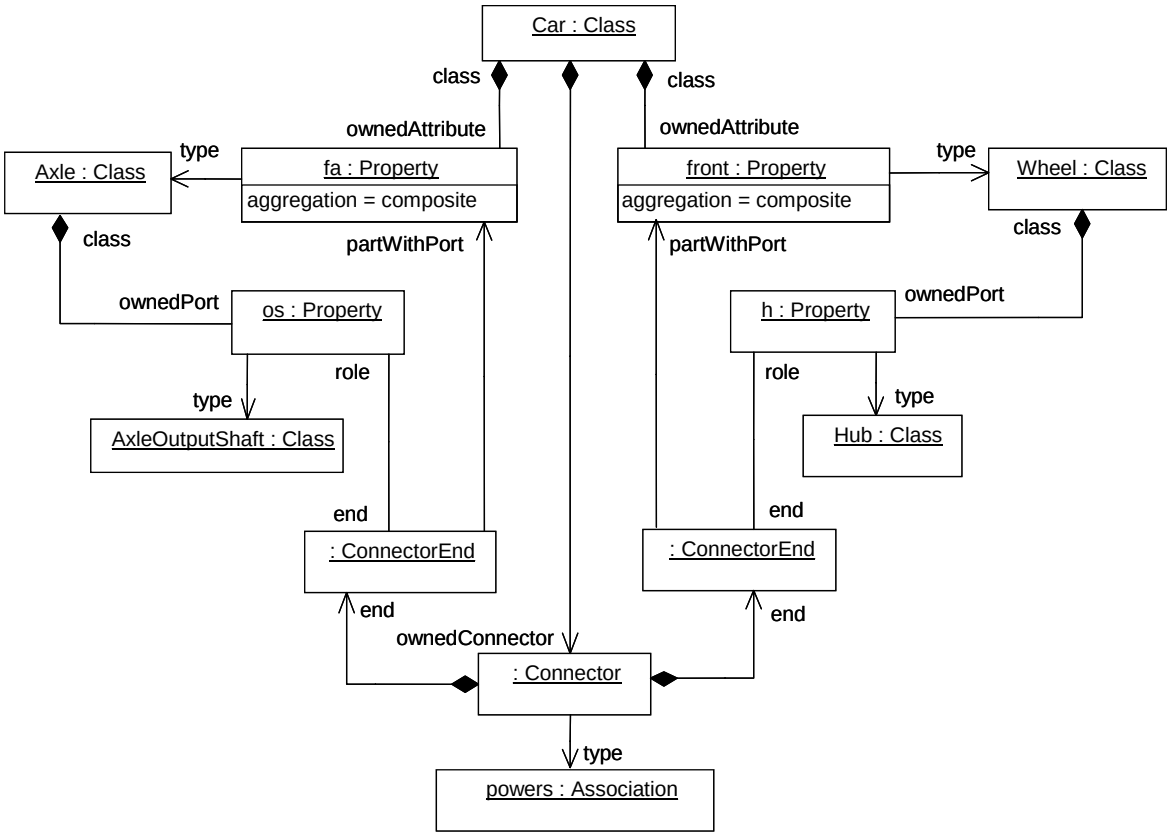


Figure 16: Repository Model for Portion of Figure 14

[18] The UML 2 metamodel does not support navigation to levels deeper than two. This is addressed in an extension of UML 2 for systems engineering [10] by extending connector ends to refer to an ordered list of properties of unlimited length. However, two levels of navigation are usually enough if associations are modeled as having composite structure [11]. Unfortunately, the UML 2 metamodel does not support this either, because it does not model navigation from links (instances of associations) to the objects at the end of the links as OWNEDATTRIBUTE. Structured associations are useful in many applications, for example, phones communicating over a complex network of objects and associations that are contained in a single association between the phones. They also enable a unification of objects and associations based on how many contained connectors refer outside the composite, zero for objects, more than zero for associations [11]. This will be addressed in a revision of UML or in the systems engineering extension.

6 MESSAGE-FORWARDING ACROSS PORTS

Ports can be used primarily for communication, rather than linking objects, as shown in Figure 17. Connectors define the channels along which messages may be sent[19]. Messages sent to a port from outside a composite object are forwarded to the composite object or to internal objects, based on how the port is connected to them, while messages sent to the port from the inside are forwarded to the connected external objects. For example, in Figure 17, in an instance of FTPSESSION (which only exists logically), a link is created between ports of the computers playing the VISITOR and HOST parts. The visitor sends requests for gets and puts to its FTPIN port, which are forwarded to the FTPOUT port at the host computer on the other end of the session link, then to the FTPHOST process. Likewise, replies from the host are sent from the FTPHOST process to the host port, forwarded across the link to the visitor port, and then to the FTPVISITOR process.

When the session is closed, the host and visitor process instances are destroyed, along with the links to their ports, and the session instance is destroyed, which destroys the connection link between the computers. Destruction propagation does not apply to the visitor and host computers, because they are not strongly composed by the session, as indicated by the dashed lines (see earlier discussion on destruction propagation). Message forwarding stops when a connector reaches a port that has no connector on the other side leading to an internal part. Messages to this kind of port, called behavior ports, are handled by the object owning the port[20].

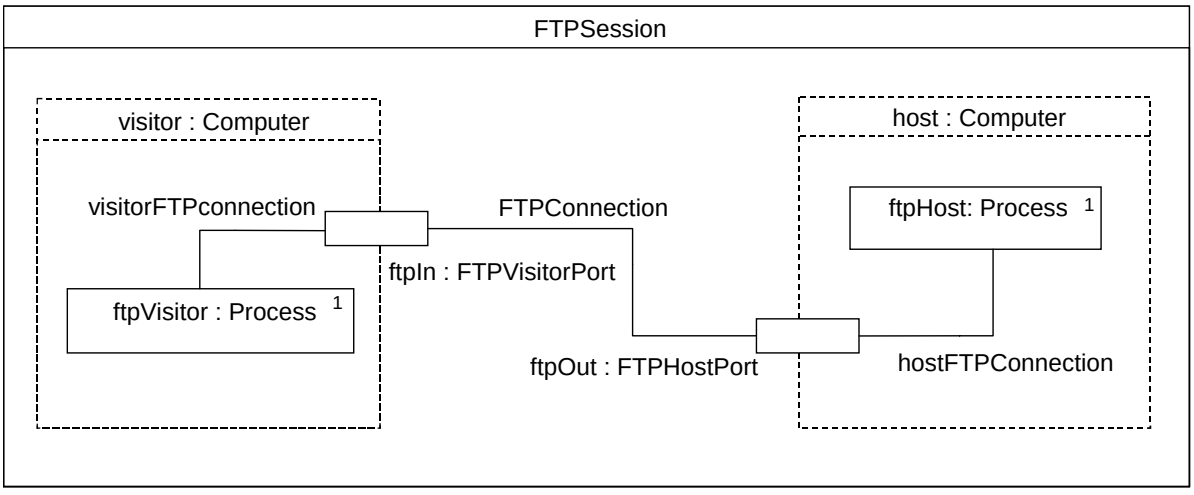


Figure 17: Ports for Message-passing

[19] This generalizes the usual object-oriented approach that objects can only communicate with other objects they are directly related to.

[20] This can be notated explicitly with an internal connector from the behavior port to a small rounded rectangle in the owning class.

Messages sent to a port must be supported either by:

- objects that are values of the port. These are called *port objects*.
- the object owning the port, if there is no value for the port.
- or both of the above if the port object forwards messages to the owning object.

Messages are either:

- calls to operations, which are usually dispatched to methods on the receiving object.
- or sending of signals, which are buffered on the receiving object and handled by behaviors in the objects.

The messages an object sends and receives are declared on its class (its type). This applies to both composite objects and port objects. Specifically, messages are declared on interfaces supported by the class, as operations and signal receptions, collectively called behavioral features. Since messages might be sent across a port from outside or inside, a class can declare its interfaces as provided or required, which are notated in various ways as shown in the class diagram fragments in Figure 18 and Figure 19. Provided interfaces declare messages sent from outside the containing object. These are modeled with a realization dependency from the class to the interface. This is same model used for conventional interfaces supported by classes. Notations for provided interfaces include a dashed line with hollow arrowhead, or a dashed line with stick arrowhead and «realize» keyword. The "lollipop" or "ball" notation option can be used on port classes, or directly on ports shown on the owning class, COMPUTER in this example, as shown in Figure 19. Required interfaces declare the messages sent from inside the composite object. These are modeled with a uses dependency from the class to the interface. One notation for this is a dashed line with stick arrowhead and «uses» keyword. Another is the "socket" notation on port classes, or directly on the ports notated on the owning class[21]. Provided and required interfaces are modeled this way to support the same interface provided by one port and required by another.

[21] Ball and socket notations are useful on the composite structure diagram, attached to port symbols, even though UML does not define this presentation option explicitly. The ball and socket can even be joined as a substitute notation for a connector, although this requires the connections to be shown twice in some cases, one of each direction of communication. Some might find the ball and socket notation confusing, since the ball indicates incoming messages and the socket indicates outgoing messages.

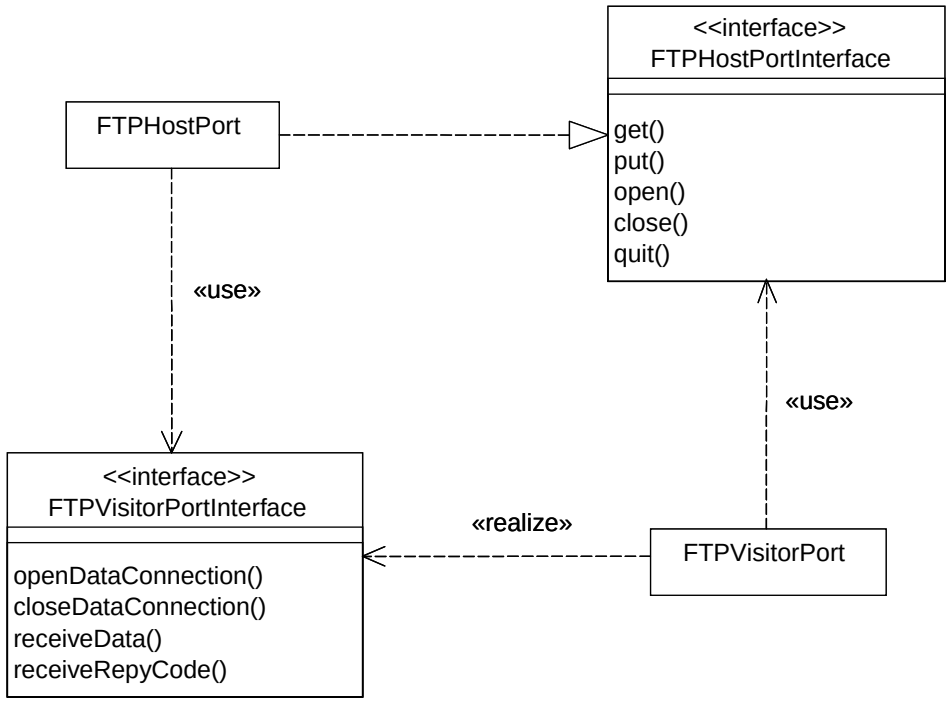


Figure 18: Port Interfaces

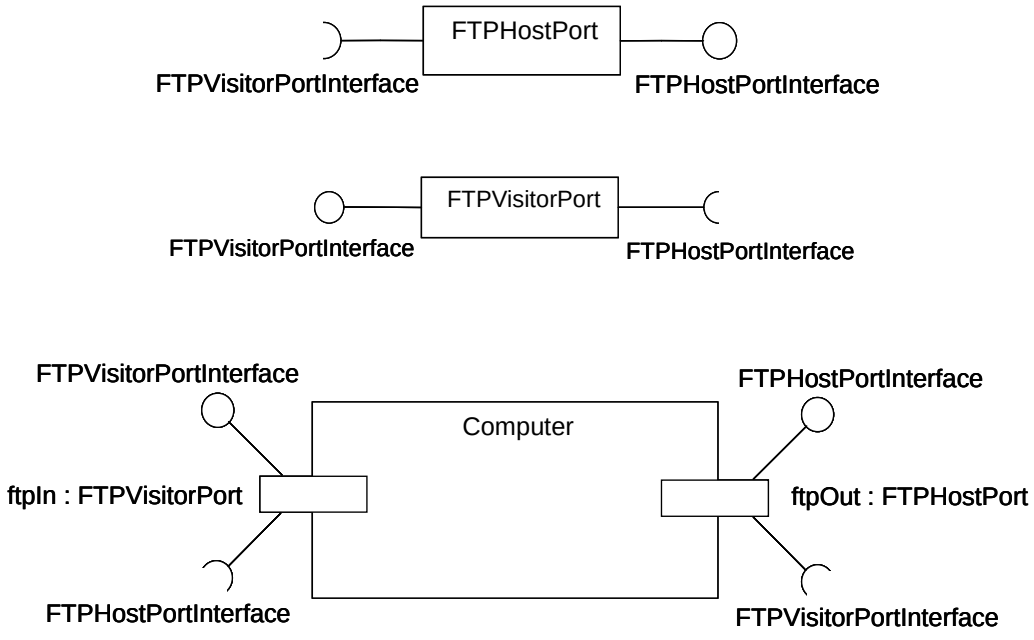


Figure 19: Other Notations for Figure 18

Provided and required interfaces are defined with respect to messages arriving from outside a composite class or sent to the outside, but they also declare capabilities for messages arriving from the inside and sent to the inside. For example, in Figure 17, messages sent from the object playing the FTPVISITOR part to the port object of FTPIN are operations such as GET and PUT. These are defined in the required interface of FTPIN, FTPHOSTPORTINTERFACE, not the provided interface FTPVISITORPORTINTERFACE, even though the messages come into the port object of FTPIN. Normally messages sent to an object are defined in the provided interface, but port objects are treated specially in this regard. Similarly, messages sent from the FTPIN port to the FTPVISITOR part are operations such as OPENDATACONNECTION and RECEIVEDATA. These are defined in the provided interface of FTPIN, FTPVISITORPORTINTERFACE, not the required interface FTPHOSTPORTINTERFACE. Normally, messages sent out from an object are defined in the required interface, but again port objects are treated specially in this case.

Ports can be implemented in a lightweight fashion, as pure interaction points, with no port object created at all at runtime. In these applications, messages are sent to the object owning the port with instruction that they are sent "through" a particular port[22]. For example, an invocation action in the host object of Figure 17 can have the host object itself as a target, with the instruction that the message is sent out through the FTPOUT port, to be forwarded to the visiting computer. A model compiler can calculate the ultimate targets for forwarding, and translate these actions to send to those targets. For example, in Figure 17, an invocation action in the HOST object that targets the host itself though the FTPOUT port can be translated to send messages to the object playing the FTPVISITOR part of the visitor object directly. Compilation is necessary in lightweight implementations, because there are no port objects at runtime that can be linked together and determine forwarding targets dynamically.

Port classes, such as FTPVISITORPORT and FTPVISITORHOST, are still needed in lightweight implementations as the type of port properties, because the dependencies of port classes determine which interfaces are provided and which are required. Port classes are not instantiated in the lightweight implementation, however, so the lower multiplicity of the port property must be zero[23]. Since port classes are not instantiated, the connector has no association to instantiate either, and there is no association needed between the port classes or interfaces (compare to Figure 20).

Ports can also be implemented in a heavyweight way with port objects created at runtime, to provide dynamic services. For these applications, invocation actions [9] call operations and send signals directly to port objects. When a port object receives a message, it can perform various functions, such as filtering messages, modifying them, or routing them dynamically, and can also keep state, for example, to count the number of messages passing through it. Messages can still be sent in the lightweight style to objects owning heavyweight ports, with instruction to go through the port, but the previously mentioned compilation cannot be performed, because the heavyweight port might not determine how to forward the message until runtime. The lightweight style of message

[22] UML extends its invocation actions [9] to support sending messages through ports.
[23] It would be useful to set the upper multiplicity to zero also, but UML 2 does not support this currently.

passing is best in general, however, because it works regardless of whether the targeted port is heavy or light, as long as the model compiler is designed to tell the difference. Heavyweight ports can still implement forwarding as it would have been with the lightweight implementation, but the inefficiency of forwarding cannot be compiled away without special directives to the compiler.

Port interfaces in heavyweight implementations are related by the same association that the connectors refer to, for example FTPCONNECTION, as shown in Figure 20. Each port class supports navigating associations to objects supporting the other end's interface. For example, the instances of the port class FTPHOSTPORT support navigating along the visitor association end to objects supporting the FTPVISITORPORTINTERFACE.

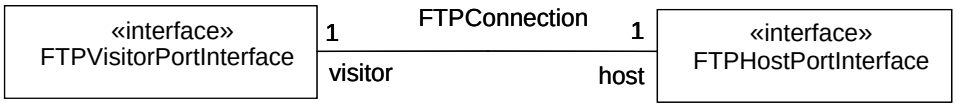


Figure 20: Association between Port Interfaces

Ports and message forwarding provide better insulation of message source and targets than typical object and component approaches, especially in lightweight implementations, which do not require associations between assembled subcomponents. Conventional approaches only support provided interfaces, so the sender must identify a target object directly, often by an explicit link to it, even to use its interfaces. The ports approach enables the sender to declare required interfaces, and to send messages to its own ports when communicating with the environment, rather than identifying an external target object directly. Classes defined this way are assembled by wiring them together with connectors. The resulting composite class is responsible for creating instances of its constituents, and either linking them together according to connectors, or enabling message compilation, depending on the implementation. Classes and components are defined more independently of their usage, providing better plug-compatibility[24].

7 INHERITANCE OF COMPOSITE STRUCTURE

Connectors and association ends, including parts, inherit from class to subclass as all features do. Subclasses can specialize inherited parts and connectors, and introduce new ones. Figure 4 and Figure 5 show inheritance of composite association ends (parts) and specializing them, in particular, the association ends POWERSOURCE and POWERTRANSMITTER. Figure 21 is a combined class and composite structure diagram showing inheritance[25], and specialization and addition of parts. The parts of the general

[24] UML 2 defines further refinements and applications of composite structure for components and collaboration not covered in this article.

[25] This presentation option is not explicitly defined in UML, but it is useful for explanation of concepts and is a natural extension of defined diagrams.

structure defined by VEHICLE are specialized in ROADVEHICLE and BOAT, for example, the type of the VEHICLEFRAME association end is specialized from the class VEHICLEFRAME to CHASSIS and HULL (the redefinitions on parts, such as `{redefines vehicleFrame}`, are omitted for brevity). The inheritance continues down to TRUCK, which adds the part TRAILER:TRAILER and the :PULLS connector, specializes the type of POWERSOURCE, and the multiplicity of TRAILER. The class diagram for the part types is shown in Figure 22.

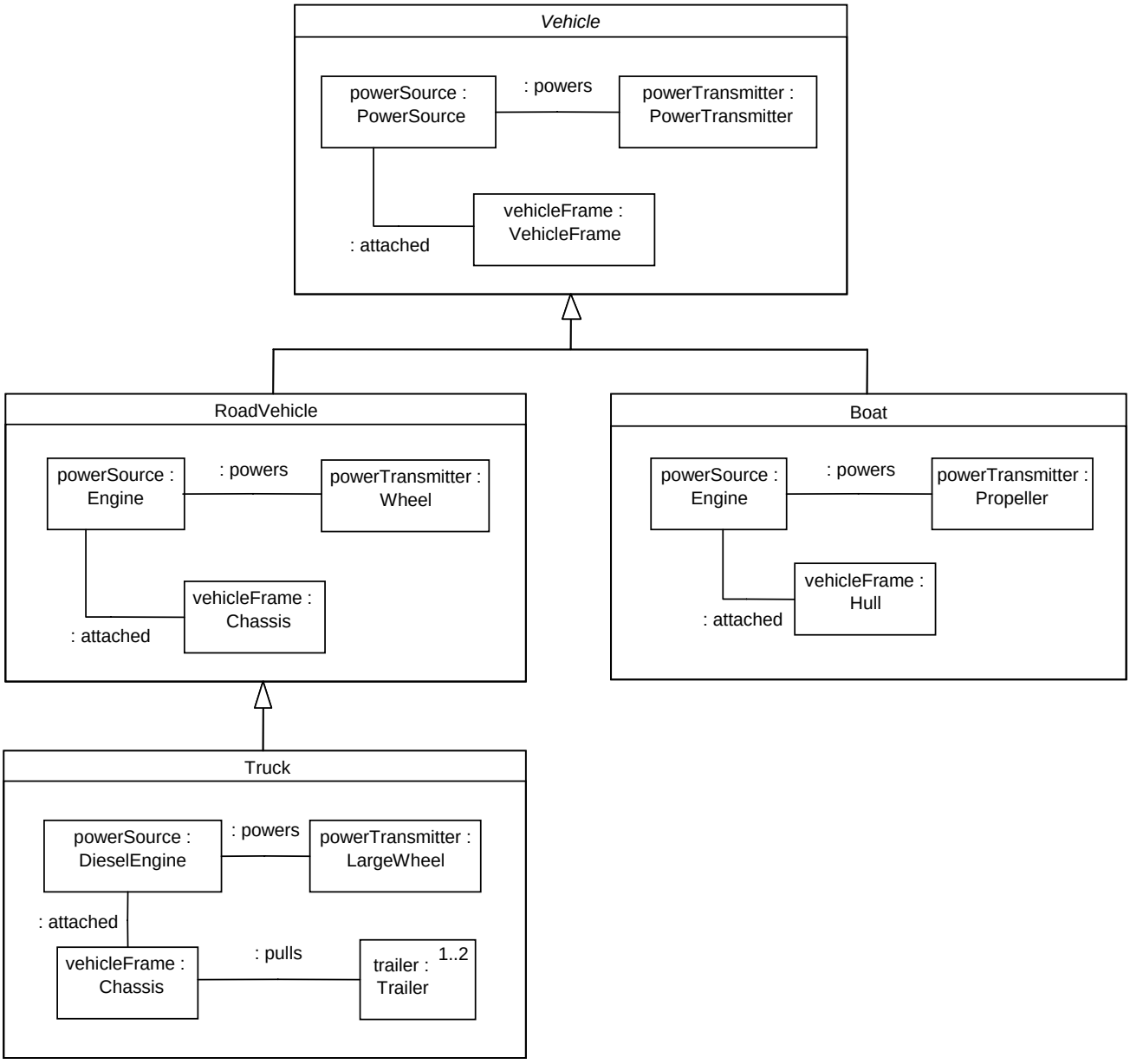


Figure 21: Inheritance of Composite Structure (combined class/composite structure diagram)

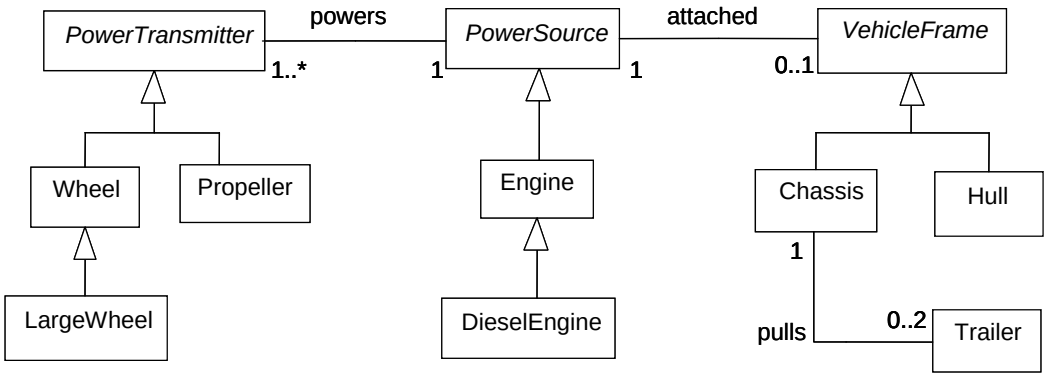


Figure 22: Clases and Associations Used in Figure 21

8 CONCLUSION

This article covers the updated composition model in UML 2. It is first shown that conventional classes and associations are not sufficient for simple examples of composition. Earlier versions of UML did not properly support associations between classes at the same level of decomposition, allowing a number of undesired links to be created at runtime, sometimes between the contents of unrelated composite objects. The workarounds necessary to correct these problems require complex diagrams with many specialized classes and associations.

The UML 2 composition model resolves these issues by recognizing that “parts” of an object are best identified by navigating from the containing object along association ends or attributes to the contained objects. Links between parts are modeled as connections between association ends or attributes of the containing object. This approach enables a class to model a network of objects playing the various parts defined by the class. Objects can play specific parts at some times and not others, with the links to other objects in the composite maintained automatically. Since the network structure is abstracted into a class, it is inherited to subclasses, where new connections and parts can be added and specialized.

The model also supports connections between ports, which are accessible parts of parts. This requires double chains of navigation from the overall containing object to identify the proper objects to link at runtime. These links or connections can be used to forward messages sent from the internals of one object to the internals of another, within the context of an overall composite structure. The internal behaviors of an object can send messages to its own ports, which are forwarded along links or connections specified by the composite structure, eventually reaching the internal behaviors of other objects. This provides significantly better encapsulation and plug-compatibility of system components than conventional techniques.

ACKNOWLEDGEMENTS

Thanks to Evan Wallace, James Odell, Steve Cook, Bran Selic, and Doug Tolbert for their input to this article, to Robert Thompson for assistance with the FTP example, and to Michael Williams for the original Car/Boat example.

REFERENCES

[1] Object Management Group, "UML 2.0 Superstructure Specification," http://www.omg.org/cgi-bin/doc?ptc/04-10-02, October 2004.

[2] Object Management Group, "OMG Unified Modeling Language," version 1.5, http://www.omg.org/cgi-bin/doc?formal/03-03-01, March 2003.

[3] Cook, S., Kent, S., Software Factories, Appendix B, Wiley, 2004.

[4] Odell, J., "Six Different Kinds of Composition," in *Journal of Object-Oriented Programming*, vol. 5, no. 8, January 1994.

[5] Bock, C., Odell, J., "A Foundation for Composition," in *Journal of Object-Oriented Programming*, vol. 7, no. 6, pp. 10-14, October 1994.

[6] Mishelevich, D., et. al., "Representing Composites with Valueclass Enhancements and the Relation Form of Recursive Units," in New Generation Knowledge System Development Tools, Phase 2 Interim Report, DARPA Contract F30602-85-C-0065, May 1988.

[7] Bock, C., Odell, J., "A More Complete Model of Relations and Their Implementation, Part III: Roles," *Journal of Object-Oriented Programming*, vol. 11, no. 2, pp. 51-54, May 1998.

[8] Object Management Group, "UML 2.0 OCL Specification," http://www.omg.org/cgi-bin/doc?ptc/03-10-14, October 2003.

[9] Bock, C., "UML 2 Activity and Action Models, Part 2: Actions," in *Journal of Object Technology*, vol. 2, no. 5, pp. 41-56, September-October 2003, http://www.jot.fm/issues/issue_2003_09/column4.

[10] SysML Partners, “Systems Modeling Language: SysML,” http://www.omg.org/cgi-bin/doc?ad/04-10-02, August 2003.

[11] Bock, C., Odell, J., "A More Complete Model of Relations and Their Implementation Part IV: Aggregation,” *Journal of Object-Oriented Programming*, vol. 11, no. 5, pp. 68-70, 85, September 1998.

About the author

Conrad Bock is a Computer Scientist at the U.S. National Institute of Standards and Technology, specializing in process models and ontologies. He is responsible for several implementations of composition in frame-based systems, applied in a variety of domains, and contributed to the development of the UML 2 composition model. He can be reached at conrad.bock at nist.gov.