Abstract Thought in Object Oriented Software Development

What does abstraction mean, how can it be applied in Object Oriented development to improve software systems?

© Martin Hunter, July 2006

1 Executive Summary

Abstraction is a core concept in Object Oriented (OO) development. This paper discusses the nature of abstraction, and introduces and compares some of the key mechanisms that OO technologies incorporate to support it. We conclude with some useful rules to help you "think abstract" and incorporate abstract principles into your software design and development processes.

2 The Concept of Abstraction

According to Wikipedia "abstraction is the process of reducing the information content of a concept, typically in order to retain only information which relevant for a particular purpose". In short, then, abstraction is the *simplification* of a concept in a specific *direction*. Abstraction removes given specific concept details to leave a more general concept in which that detail is ambiguous.

Consider the statement that "I saw many different *cars* driving on the *road* today". The identification of a specific object as being a *car* and the "driving on" relationship that such objects have with *roads* are both abstractions of those objects.

3 Abstraction in Object Orientation

In computer science, the term *abstraction* has a specific connotation: It is a mechanism by which details are factored out of broader concepts to allow us to focus in on specific concepts. Abstraction is a central concept of the OO software design paradigm in which specific *types* of objects can be *generalized* (abstracted) to less specific types of objects.

3.1 Abstraction through Inheritance

This logic is embedded in the OO principle of *inheritance*, in which one type (class) is a *specialization* of (inherits from or extends) another type (class). In OO inheritance, the generalized type in the relationship is usually referred to as the *super-class* and the specialized type the *sub-class*. The following UML class diagram illustrates the relationship, in which the solid-head arrow specifies that *super* generalizes *sub* (or, inversely, that *sub* is a specialization of *super*):

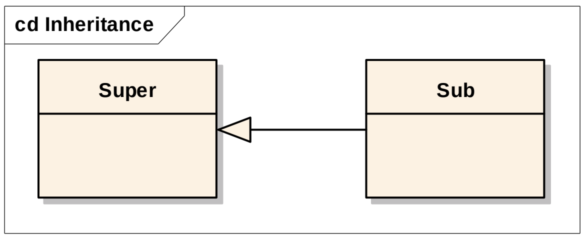

Figure 1: Class Inheritance

In OO, abstraction morphs two broader concepts of *control abstraction* and *data abstraction*. Control abstraction relates to the operation of an object from a functional viewpoint, whilst data abstraction relates to the information about the object or that the object holds. As such, when a sub class inherits from a super class in OO, both the *methods* (operational aspects) and the *data* (information attributes) of the super class are inherited by the sub class:

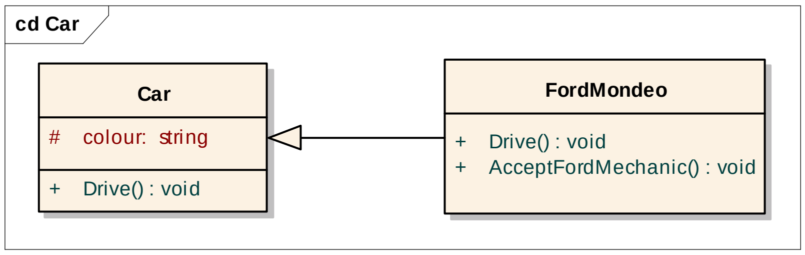

Figure 2: Specialisation of data and functionality

The above diagram illustrates a scenario in which the sub class "Ford Mondeo" inherits *all* of the methods and data from super class "Car" and supplements this with it's own operations and data. In other words, a Car is a generalization, an *abstraction*, of a Ford Mondeo.

Abstract Classes

In OO, the term *abstract class* has a specific meaning: An *abstract class* is one which cannot, in itself, be instantiated. Objects can not be constructed from an abstract class. The purpose of an abstract class is, therefore, to allow the abstraction of higher level concepts away from "real" classes to afford an appropriate logical grouping of "real" classes and to facilitate their inheritance of concept-specific methodological and data sets.

Referring back to our previous example of the "Ford Mondeo" and "Car" classes, we can assert that a Ford Mondeo *can* be instantiated, because Ford Mondeo type objects are *real* objects. We *cannot*, however, instantiate a Car per se, because there is simply no such thing as a "Car" type object in real life. In this way, we can assert that Car is an *abstract super type* of Ford Mondeo.

This relationship would be represented in UML as follows. Note that the name of the Car class and the Drive() methods have been italicized to indicate that they are abstract:

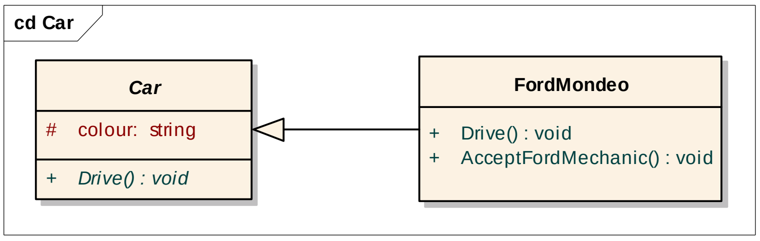

Figure 3: Abstract Classes

Interfaces

In OO, *interfaces* are very similar to abstract classes, except that an interface cannot contain any implementation detail where an abstract class can. Interfaces, therefore, define only the *operational* aspects of an abstract type: Interfaces can contain *methods definitions*, but not the implementation of those methods or any data about the type.

Many people find it helpful to think of an interface type as *contract* that an *implementing class* (a sub class that inherits from the interface) needs to adhere to. This manner of thinking is useful because interfaces enforce a *requirement* for classes that inherit from them to *implement* a given set of *operations*. In this way, implementing an interface can be seen as a contractual obligation for a class to offer a certain set of functionality.

In general, a class that inherits from an interface is said to be a *realization* of that interface. This terminology positions interfaces squarely as a functional abstraction of classes: The class is something that makes the interface "real".

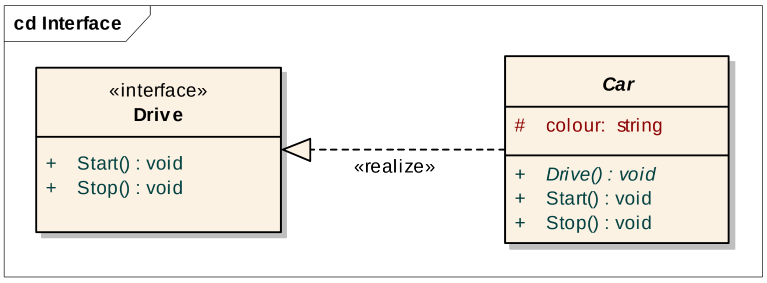

Figure 4: Interfaces

The above diagram illustrates the UML syntax for the realization of an interface by a class. The Car class implements the Drive interface, an entirely *functional* abstraction of Car that incorporates operations relating to *driving*.

3.2 <u>Abstraction through Composition</u>

Inheritance is one way in which abstractions can be defined in OO. Another way is *composition*. Composition relates to the combining of several objects to form another object. Unlike inheritance, abstraction through composition is not about leaving information out to simplify things. It is about *breaking larger things up* into smaller bits, each having a simpler and more focused purpose and use.

To understand composition, let look at an example. A Car object might be composed of many other objects such as Wheels, Engines, Doors, Windows and so on. Each of the objects that the Car is composed of are independent objects in their own right: They can be removed from the Car and still exist. The following UML class diagram illustrates, where the composition relationship between objects is denoted by a white diamond:

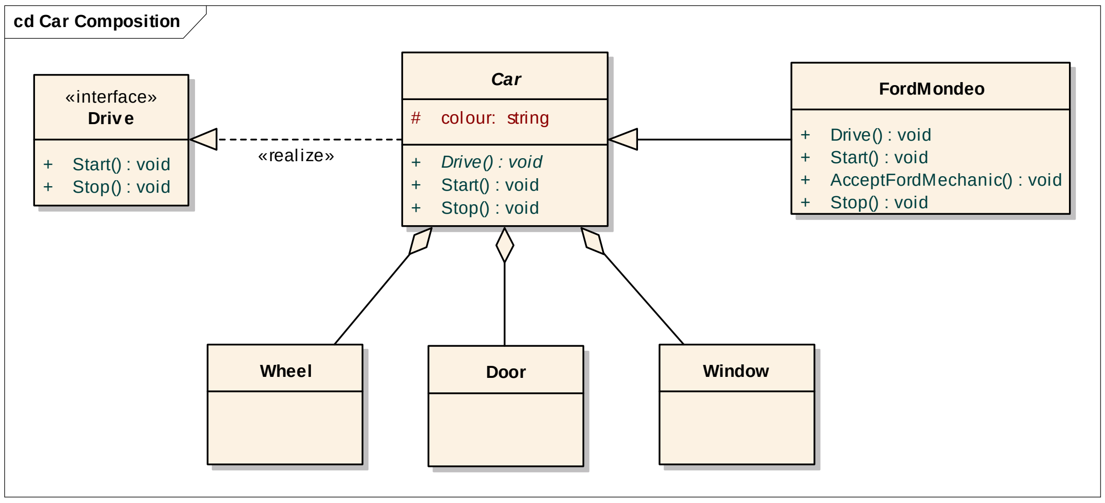

Figure 5: Shared aggregation

This kind of composition is called a *shared* (or *weak*) aggregation. The Car type is an *aggregate* of other types (such as Wheel), and each of those other types can exist *outside of the Car*. In other words, they can be *shared* with other things that might be composed of them (an instance of Wheel, for instance, might be removed from a Car and attached instead to a Wheel Barrow, though clearly the Wheel cannot be attached to both simultaneously!). Shared aggregations are commonly identified by a "has-a" relationship between objects: A Car *has a* Wheel (or probably more like four Wheels, actually).

Composite (or *strong*) *aggregation* relates to situations where an object is composed of objects that *cannot exist outside the context of that object.* For instance, a Car might also have a Registration Number, Year of Manufacture and so on. These objects are Car-object specific, and cannot exist outside the Car: One couldn't remove the Year of Manufacture object from a Car and attach it to something else. Composite aggregations are commonly identified by a "contains a" relationship between objects, and are represented in UML by a black diamond:

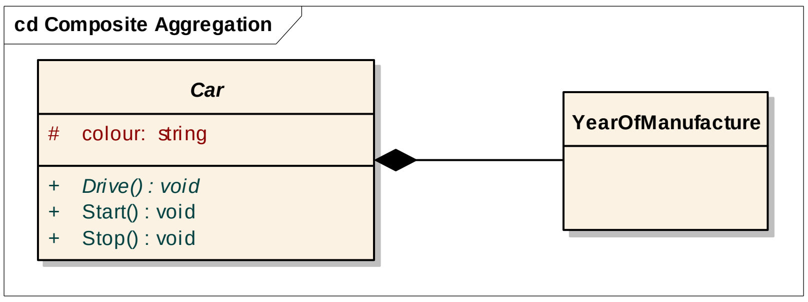

Figure 6: Composite aggregation

3.3 <u>Comparing Abstraction through Inheritance and Composition</u>

Varied problem contexts often call for equally varied solutions; one size does not fit all. This is certainly true with the problem of identifying whether to use composition or inheritance in OO developments.

The Gang of Four suggest that composition should always be favoured over inheritance. This is largely because composition promotes the *independence* of classes. Composition based designs involve autonomous units of functionality which can easily be decomposed, recomposed and passed around. This promotes **reuse**.

Inheritance, on the other hand, promotes *dependence* between classes. Inheritance yields tight relationships between super and sub types, thereby making it more difficult (or impossible) to untangle relationships. Perversely, inheritance binds different classes together and reduces the available scope for reuse.

4 The Abstract Thought Process

As stated earlier, abstraction is a *process* of concept simplification. Whether by inheritance of composition, abstraction in OO development is **good** because it facilitates the separation of concerns and thus promotes the autonomy of types in your system.

Looking to our earlier examples, imagine what a system *without* the Car class would be like. How would you know that a Ford Mondeo and a VW Polo were both types of Car? How would you be able to stipulate that both had to implement the *Drive* interface? How would you be able to pass a Ford Mondeo or a VW Polo to a Road class and not have to implement logic in the Road class that tells it to treat them the same way? Clearly the Car class is an important abstraction for what would otherwise be a much more complex system to develop and use.

Abstraction, then, is an important ingredient to OO systems. But how do we go about the process of abstraction? What rules can we cite that will help us develop systems with appropriate levels of abstraction amongst types?

Spotting opportunities for abstraction in software design is a skill. As you become more experienced, you are likely to become more proficient at building systems that incorporate appropriate levels of abstraction, in the right ways and in the right places. In general, I would encourage you to "think abstract" when you design and develop software. The following rules have been assembled to help you along the way:

1. Consider the purpose of things

All components of a well developed software system have a purpose and a design, whether those things are documented, understood or otherwise. When you're designing an OO system, take time to consider the *purpose* of the classes and components you are designing. Would it be logical to abstract different bits of functionality and data out into other types? Are there any not-so-obvious abstractions that can be made which would improve flexibility and extensibility of the design significantly?

2. Don't be lazy

Whilst it's tempting to use inheritance as a means of cloning useful code from other classes, it's not intended for this purpose. If you find yourself doing this, then consider whether composition might help. Can the functionality you want to inherit be abstracted logically into an independent class which can then be invoked by other classes?

3. Think ahead

When you think of abstraction, consider what might happen in future. Would it be useful to abstract now to future proof my design? Is it likely that it will help in future? Don’t be swayed by developers who believe future proofing is a costly exercise with little pay-off. In my experience, future proofing your design is an excellent idea because, more often than not, the cost of abstraction now is insignificant in comparison to the need to retro-fit abstraction once a significant code base is established. Remember, if in doubt, abstract!

4. Don’t be afraid to break convention

Just because you’ve seen it done one way a hundred times before doesn’t mean to say it’s the best way out there. If you think abstraction will help you, it probably will. Use abstraction to abbreviate the design of third party classes and libraries if necessary to help separate concerns and decouple code bases.