

Chapter 2

Basic Principles of the Object-Oriented Paradigm

2.1 Abstraction

One of the most appreciated advantages of object-oriented versus other modern programming paradigms is the direct support for each of the most important and used principles of abstraction. The Dictionary of the Object Technology defines abstraction as “Any model that includes the most important, essential, or distinguishing aspects of something while suppressing or ignoring less important, immaterial, or diversionary details. The result of removing distinctions so as to emphasize commonalities.” Abstraction is an effective way to manage complexity as it allows for concentrating on relevant characteristics of a problem. Abstraction is a very relative notion; it is domain and perspective dependent. The same characteristics can be relevant in a particular context and irrelevant in another one.

The abstraction principles used in the object-oriented approach are classification/instantiation, aggregation/decomposition, generalization/specialization, and grouping/individualization. By providing support for the abstraction principles, the object-oriented paradigm makes it possible to use conceptual modeling as an efficient tool during the phases of analysis and design. Conceptual modeling can be defined as the process of organizing our knowledge of an application domain into hierarchical rankings or orderings of abstraction, in order to better understand the problem in study [94].

Classification is considered to be the most important abstraction principle. It consists of depicting from the problem domain things that have similarities and grouping them into categories or classes. Things that fall into a class/category have in common properties that do not change over time. *Instantiation* is the reverse operation of *classification*. It consists of creating individual instances that will fulfill the descriptions of their categories or classes. The majority of object-oriented languages provide capabilities for creating instances of classes/categories.

Figure 2.1 shows an example of *classification* and *instantiation*. Concept *Tractor* represents a set of properties that are typical for a tractor, regardless of their brand, horse power, etc. Therefore, concept *Tractor* represents a classification.

Fig. 2.1 Examples of classification and instantiation

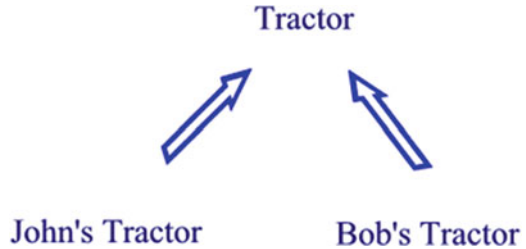
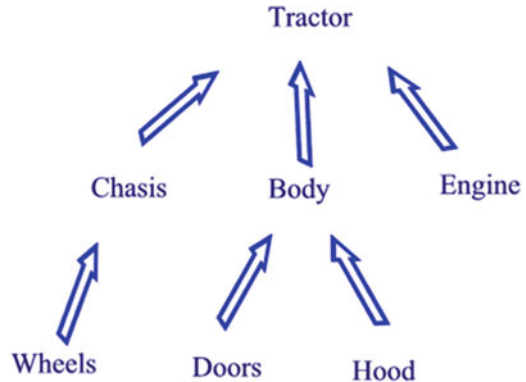


Fig. 2.2 Examples of aggregation and decomposition



Bob's *Tractor* is a particular tractor that has some particular properties, the most important being that it is Bob's property. Therefore, concept Bob's *Tractor* represents an instantiation.

The second abstraction principle is *aggregation*. *Aggregation* refers to the principle that considers things in terms of part-whole hierarchies. Concepts in a problem domain can be treated as aggregates (i.e., composed of other concepts/parts). A part itself can be considered as composed of other parts of smaller granularity. *Decomposition* is the reverse operation of *aggregation*; it consists of identifying parts of an aggregation. Object-oriented languages provide support for aggregation/decomposition by allowing objects to have attributes that are objects themselves. Thus, complex structures can be obtained by using the principle of aggregation. Note that some authors use the term *composition* instead of *aggregation*.

Figure 2.2 shows an example of *aggregation* and *decomposition*. Concept *Tractor* can be considered as an aggregation/composition of other concepts such as *Chassis*, *Body*, and *Engine*. Concept *Body* can be considered as one of the parts composing a more complex concept such as *Tractor*.

The third abstraction principle is *generalization*. *Generalization* refers to the principle that considers construction of concepts by generalizing similarities existing in other concepts in the problem domain. Based on one or more given classes, *generalization* provides the description of more general classes that capture the common similarities of given classes. *Specialization* is the reverse operation of *generalization*. A concept A is a specialization of another concept B if A is similar to B and A provides some additional properties not defined in B.

Fig. 2.3 Examples of generalization and specialization

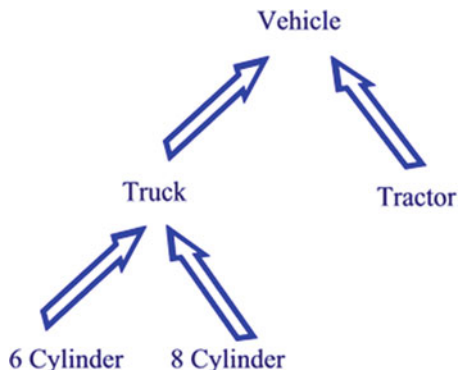
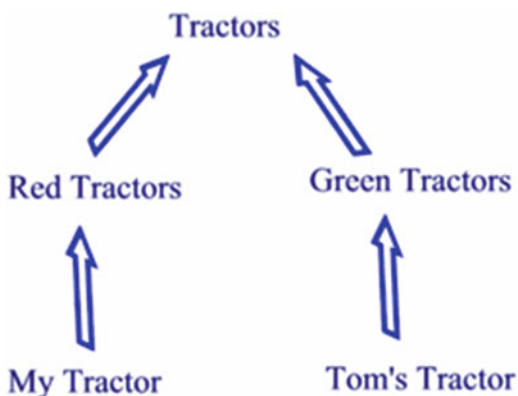


Fig. 2.4 Examples of grouping and individualization



Object-oriented languages provide support for generalization/specialization as they allow for creating subclasses of existing classes and/or creating more general classes (superclasses) of existing classes. Creating a subclass of an existing class corresponds to specialization and creating a superclass of an existing class corresponds to generalization. It is important to note that concept A is a *generalization* of concept B if and only if B is a *specialization* of concept A [79]. Figure 2.3 shows an example of generalization and specialization.

Concept *Truck* is a specialization of concept *Vehicle*. This is because *Truck* has all the properties of concept *Vehicle* and some additional ones that make it a special *Vehicle*. In reverse, concept *Vehicle* is a generalization of concept *Truck* as all trucks are vehicles.

The fourth abstraction and perhaps the least obvious is *grouping* [94]. In conceptual modeling, often a group of concepts needs to be considered as a whole, not because they have similarities but because it is important that they be together for different reasons. Object-oriented languages provide a mechanism for grouping concepts together such as sets, bags, lists, and dictionaries. *Individualization* is the reverse operation of grouping. It consists of identifying an individual concept selected among other concepts in a group. Individualization is not as well established as a form of abstraction [94]. Figure 2.4 shows an example of grouping and individualization.

All tractors used in a farm can be grouped in one category regardless of their brand, color, horsepower, and year of production and be represented by one concept such as *Tractors*. In case we need to use one of them with a certain horsepower, then we need to browse the set of tractors and find that particular individual that satisfies our needs. In this case, we have individualized one element of the set based on some particular criterion. When we say *Tom's Tractor*, we have used the ownership as criterion for individualizing one of the tractors, the one that belongs to Tom.

2.2 Encapsulation

The Dictionary of the Object Technology defines encapsulation as “The physical location of features (properties, behaviors) into a single black box abstraction that hides their implementation behind a public interface.”

Often, encapsulation is referred to as “information hiding.” An object “hides” the implementation of its behavior behind its interface or its “public face.” Other objects can use its behavior without having detailed knowledge of its implementation. Objects know only the kind of operations they can request other objects to perform. This allows software designers to abstract from irrelevant details and concentrate on what objects will perform.

An important advantage of encapsulation is the elimination of direct dependencies on a particular implementation of an object's behavior. The object is known from its interface, and clients can use the object's behavior by only having knowledge of its interface; the particular implementation of an object's interface is not important. Therefore, the implementation of the object's behavior can change any time without affecting the object's use. Encapsulation helps manage complexity by identifying a coherent part of this complexity and assigning it to individual objects.

The fact that an object “hides” the implementation of its behavior by exposing only its “public face” could be beneficial to other objects that need its behavior. The “interested” objects could consider more than one option while looking for a specific functionality that satisfies their needs. They need only to “examine” the interfaces of candidate objects. Objects with similar behavior could serve as substitute to each other.

2.3 Modularity

The Dictionary of the Object Technology defines modularity as “The logical and physical decomposition of things (e.g. responsibilities and software) into small, simple groupings (e.g., requirements and classes, respectively), which increase the achievements of software-engineering goals.”

Modularity is another way of managing complexity by dividing large and complex systems into smaller and manageable pieces. A software designing method is modular if it allows designers to produce software systems by using independent elements connected by a coherent, simple structure. Meyer [64] defines a software construction method to be modular if it satisfies the five criteria:

Modular Decomposability: a software construction method satisfies modular decomposability if it helps in the task of decomposing a software problem into a small number of less complex subproblems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them.

Modular Composability: a software construction method satisfies modular composability if it favors the production of software elements which may then be freely combined with each other to produce new systems, possible in an environment quite different from the one in which they were initially developed.

Modular Understandability: a software construction method satisfies modular understandability if it helps produce software in which each module can be understood without having to examine other interrelated modules.

Modular Continuity: a software construction method satisfies modular continuity if a small change in the requirements will impact just one or a small number of modules.

Modular Protection: a software construction method satisfies modular protection if the effect of an exception occurring at run time will impact only the corresponding module or a few neighboring modules.

The concept of modularity and the principles for developing modular software in the object-oriented approach are encapsulated in the concept of class. Classes are the building blocks in the object-oriented paradigm.

Software Engineering Techniques Applied to
Agricultural Systems
An Object-Oriented and UML Approach
Papajorgji, P.J.; Pardalos, P.
2014, XVII, 301 p. 239 illus., 89 illus. in color.,
Hardcover
ISBN: 978-1-4899-7462-4