Chapter 3

Abstract Data Types

Peter Müller
Globewide Network Academy (GNA)
pmueller@uu-gna.mit.edu

Some authors describe object-oriented programming as programming *abstract data types* and their relationships. Within this section we introduce abstract data types as a basic concept for object-orientation and we explore concepts used in the list example of the last section in more detail.

3.1 Handling Problems

The first thing with which one is confronted when writing programs is the *problem*. Typically you are confronted with "real-life" problems and you want to make life easier by providing a program for the problem. However, real-life problems are nebulous and the first thing you have to do is to try to understand the problem to separate necessary from unnecessary details: You try to obtain your own abstract view, or *model*, of the problem. This process of modeling is called *abstraction* and is illustrated in Figure 3.1.

The model defines an abstract view to the problem. This implies that the model focusses only on problem related stuff and that you try to define *properties* of the problem. These properties include

- the *data* which are affected and
- the *operations* which are identified

by the problem.

As an example consider the administration of employees in an institution. The head of the administration comes to you and ask you to create a program which allows to administer the employees. Well, this is not very specific. For example, what employee information is needed by the administration? What tasks should be allowed? Employees are real persons who can be characterized with many properties; very few are:

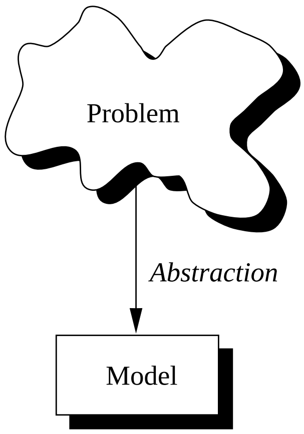

Figure 3.1: Create a model from a problem with abstraction.

- name,
- size,
- date of birth,
- shape,
- social number,
- room number,
- hair colour,
- hobbies.

Certainly not all of these properties are necessary to solve the administration problem. Only some of them are *problem specific*. Consequently you create a model of an employee for the problem. This model only implies properties which are needed to fulfill the requirements of the administration, for instance name, date of birth and social number. These properties are called the *data* of the (employee) model. Now you have described real persons with help of an abstract employee.

Of course, the pure description is not enough. There must be some operations defined with which the administration is able to handle the abstract employees. For example, there must be an operation which allows you to create a new employee once a new person enters the institution. Consequently, you have to identify the operations which should be able to be performed on an abstract employee. You also decide to allow access to the employees' data only with associated operations. This allows you to ensure that data elements are always in a proper state. For example you are able to check if a provided date is valid.

To sum up, abstraction is the structuring of a nebulous problem into well-defined entities by defining their data and operations. Consequently, these entities *combine* data and operations. They are **not** decoupled from each other.

3.2 Properties of Abstract Data Types

The example of the previous section shows, that with abstraction you create a well-defined entity which can be properly handled. These entities define the *data structure* of a set of items. For example, each administered employee has a name, date of birth and social number.

The data structure can only be accessed with defined *operations*. This set of operations is called *interface* and is *exported* by the entity. An entity with the properties just described is called an *abstract data type* (ADT).

Figure 3.2 shows an ADT which consists of an abstract data structure and operations. Only the operations are viewable from the outside and define the interface.

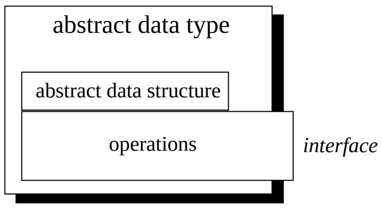

Figure 3.2: An abstract data type (ADT).

Once a new employee is "created" the data structure is filled with actual values: You now have an *instance* of an abstract employee. You can create as many instances of an abstract employee as needed to describe every real employed person.

Let's try to put the characteristics of an ADT in a more formal way:

Definition 3.2.1 (Abstract Data Type) *An* **abstract data type** *(ADT) is characterized by the following properties:*

1. *It exports a* **type**.
2. *It exports a* **set of operations**. *This set is called* **interface**.
3. *Operations of the interface are the one and only access mechanism to the type's data structure.*
4. *Axioms and preconditions define the application domain of the type.*

With the first property it is possible to create more than one instance of an ADT as exemplified with the employee example. You might also remember the list example of chapter 2. In the first version we have implemented a list as a module and were only able to use one list at a time. The second version introduces the "handle" as a reference to a "list object". From what we have learned now, the handle in conjunction with the operations defined in the list module defines an ADT *List*:

1. When we use the handle we define the corresponding variable to be of type *List*.

2. The interface to instances of type *List* is defined by the interface definition file.

3. Since the interface definition file does not include the actual representation of the handle, it cannot be modified directly.

4. The application domain is defined by the semantical meaning of provided operations. Axioms and preconditions include statements such as
   - "An empty list is a list."
   - "Let l=(d1, d2, d3, ..., dN) be a list. Then l.append(dM) results in l=(d1, d2, d3, ..., dN, dM)."
   - "The first element of a list can only be deleted if the list is not empty."

However, all of these properties are only valid due to our understanding of and our discipline in using the list module. It is in our responsibility to use instances of *List* according to these rules.

Importance of Data Structure Encapsulation

The principle of hiding the used data structure and to only provide a well-defined interface is known as *encapsulation*. Why is it so important to encapsulate the data structure?

To answer this question consider the following mathematical example where we want to define an ADT for complex numbers. For the following it is enough to know that complex numbers consists of two parts: *real part* and *imaginary part*. Both parts are represented by real numbers. Complex numbers define several operations: addition, substraction, multiplication or division to name a few. Axioms and preconditions are valid as defined by the mathematical definition of complex numbers. For example, it exists a neutral element for addition.

To represent a complex number it is necessary to define the data structure to be used by its ADT. One can think of at least two possibilities to do this:

- Both parts are stored in a two-valued array where the first value indicates the real part and the second value the imaginary part of the complex number. If x denotes the real part and y the imaginary part, you could think of accessing them via array subscription: x=c[0] and y=c[1].

- Both parts are stored in a two-valued record. If the element name of the real part is r and that of the imaginary part is i, x and y can be obtained with: x=c.r and y=c.i.

Point 3 of the ADT definition says that for each access to the data structure there must be an operation defined. The above access examples seem to contradict this requirement. Is this really true?

Let's look again at the two possibilities for representing imaginary numbers. Let's stick to the real part. In the first version, x equals c[0]. In the second version, x equals c.r. In both cases x equals "something". It is this "something" which differs from the actual data structure used. But in both cases the performed operation "equal" has the same meaning to declare x to be equal to the real part of the complex number c: both cases archieve the same semantics.

If you think of more complex operations the impact of decoupling data structures from operations becomes even more clear. For example the addition of two complex numbers requires you to perform an addition for each part. Consequently, you must access the value of each part which is different for each version. By providing an operation "add" you can *encapsulate* these details from its actual use. In an application context you simply "add two complex numbers" regardless of how this functionality is actually archieved.

Once you have created an ADT for complex numbers, say *Complex*, you can use it in the same way like well-known data types such as integers.

Let's summarize this: The separation of data structures and operations and the constraint to only access the data structure via a well-defined interface allows you to choose data structures appropriate for the application environment.

3.3 Generic Abstract Data Types

ADTs are used to define a new type from which instances can be created. As shown in the list example, sometimes these instances should operate on other data types as well. For instance, one can think of lists of apples, cars or even lists. The semantical definition of a list is always the same. Only the type of the data elements change according to what type the list should operate on.

This additional information could be specified by a *generic parameter* which is specified at instance creation time. Thus an instance of a *generic ADT* is actually an instance of a particular variant of the ADT. A list of apples can therefore be declared as follows:

```
List<Apple> listOfApples;
```

The angle brackets now enclose the data type for which a variant of the generic ADT *List* should be created. *listOfApples* offers the same interface as any other list, but operates on instances of type *Apple*.

3.4 Notation

As ADTs provide an abstract view to describe properties of sets of entities, their use is independent from a particular programming language. We therefore introduce a notation here which is adopted from [3]. Each ADT description consists of two parts:

- **Data**: This part describes the structure of the data used in the ADT in an informal way.
- **Operations**: This part describes valid operations for this ADT, hence, it describes its interface. We use the special operation **constructor** to

describe the actions which are to be performed once an entity of this ADT is created and **destructor** to describe the actions which are to be performed once an entity is destroyed. For each operation the provided *arguments* as well as *preconditions* and *postconditions* are given.

As an example the description of the ADT *Integer* is presented. Let k be an integer expression:

ADT *Integer* is

Data

A sequence of digits optionally prefixed by a plus or minus sign. We refer to this signed whole number as N.

Operations

constructor Creates a new integer.

add(k) Creates a new integer which is the sum of N and k. Consequently, the *postcondition* of this operation is $sum = N+k$. Don't confuse this with assign statements as used in programming languages! It is rather a mathematical equation which yields "true" for each value *sum*, N and k after *add* has been performed.

sub(k) Similar to *add*, this operation creates a new integer of the difference of both integer values. Therefore the postcondition for this operation is $sum = N\text{-}k$.

set(k) Set N to k. The postcondition for this operation is $N = k$.

...

end

The description above is a *specification* for the ADT *Integer*. Please notice, that we use words for names of operations such as "add". We could use the more intuitive "+" sign instead, but this may lead to some confusion: You must distinguish the operation "+" from the mathematical use of "+" in the postcondition. The name of the operation is just *syntax* whereas the *semantics* is described by the associated pre- and postconditions. However, it is always a good idea to combine both to make reading of ADT specifications easier.

Real programming languages are free to choose an arbitrary *implementation* for an ADT. For example, they might implement the operation *add* with the infix operator "+" leading to a more intuitive look for addition of integers.

3.5 Abstract Data Types and Object-Orientation

ADTs allows the creation of instances with well-defined properties and behaviour. In object-orientation ADTs are referred to as *classes*. Therefore a class defines properties of *objects* which are the instances in an object-oriented environment.

ADTs define functionality by putting main emphasis on the involved data, their structure, operations as well as axioms and preconditions. Consequently, object-oriented programming is "programming with ADTs": combining functionality of different ADTs to solve a problem. Therefore instances (objects) of ADTs (classes) are dynamically created, destroyed and used.

3.6 Excercises

1. ADT *Integer*.

   (a) Why are there no preconditions for operations *add* and *sub*?

   (b) Obviously, the ADT description of *Integer* is incomplete. Add methods *mul*, *div* and any other one. Describe their impacts by specifying pre- and postconditions.

2. Design an ADT *Fraction* which describes properties of fractions.

   (a) What data structures can be used? What are its elements?

   (b) What does the interface look like?

   (c) Name a few axioms and preconditions.

3. Describe in your own words properties of abstract data types.

4. Why is it necessary to include axioms and preconditions to the definition of an abstract data type?

5. Describe in your own words the relationship between

   - instance and abstract data type,
   - generic abstract data type and corresponding abstract data type,
   - instances of a generic abstract data type.