Abstract Data Type(ADT)

Abstract Data Type(ADT) is a data type, where only behavior is defined but not implementation.

Opposite of ADT is **Concrete Data Type (CDT**), where it contains an implementation of ADT.

Examples:

Array, List, Map, Queue, Set, Stack, Table, Tree, and Vector are ADTs. Each of these ADTs have many implementations i.e. CDT. Container is a high-level ADT of above all ADTs.

Real life example:

Book is Abstract (Telephone Book is an implementation)

ADT is to an interface (what it does) what a data structure is to a class (how it does it).

A few examples:

ADT: List

DS: ArrayList, LinkedList...

ADT: Map

DS: HashMap, TreeMap...

Abstract Data Type: ADT may be defined as a **set of data values and associated operations that are precisely specified independent of any particular implementation**. Thus an Abstract Data Type is an organized collection of information and a set of operations used to manage that information. **The set of operations defines the interface of the ADT.** As long as the ADT fulfills the conditions of the interface, it doesn't really matter how the ADT is implemented. Since, in ADT, the data values and operations are defined with mathematical precision, rather than as an implementation in a computer language, we may reason about effects of the operations, relations to other abstract data types whether a program implements the data type etc. One of the simplest abstract data type is the stack data type for which functions might

be provided to create an empty stack, to push values onto a stack and to pop values from a stack.

The basic difference between abstract data type (ADT) and concrete data type is that the latter allow us to look at the concrete representation, whereas the former hide the representation from us. An ADT may be pure ADT or Updatable ADT**. A pure ADT is one where all operations are pure functions**. This means that operations have no side effects. In particular, they do not modify or update there input arguments. They just use these arguments to generate output, which are fresh values of ADT (or of other types). Most concrete types are pure. For example, no operation on integers actually modifies an integer. Instead, all the operations like '+' produce fresh outputs.

An updatable ADT is one where some operations actually change values of the ADT. For example, suppose we had an operation called 'pop' that took a stack as an argument and modified it. ("in place", "destructively") by removing the highest priority item. This operation would be considered impure and the whole ADT would then be impure also. An ADT may be user defined ADT.

Abstract Data Type is a data type that satisfies the following two conditions:

- The representation, or definition, of the type and the operations are contained in a single syntactic unit.
- The representation of objects of the type is hidden from the program units that use the type, so only direct operations possible on those objects are those provided in the type's definition.

A user defined Abstract Data Type should provide:

- A type definition that allows program units to declare variables of the type, but hides the representation of these variables.
- A set of operations for manipulating objects of the type.

An example of a user defined abstract data type is structure. 'C' provides four basic types: int, char, float and double. However, 'C' also provides the programmer with the

ability to define his/her own types. Structure is one such example. A structure is an aggregate of different parts, where each part is of some existing type.

```
struct abc
{
int x;
float y;
};
```

The above structure definition does not create any variables, rather it creates a new type. Variables of this type may be created in a similar way to variables of a built in type.

```
struct abc a;
```

The typedef keyword allows us to create new type names for our new types.

For example:

```
typedef struct abc AB;
```

where AB is a new type name that can now be used to create new types.

```
AB b;
```

Data Structures: The following are the characteristic features of data structures:

It contains component data items, which may be atomic or another data structure (still a domain).

A set of operations on one or more of the component items.

Defines rules as to how components relates to each other and to the structure as a whole (assertions).

Data Structures:

A data structure may be static or dynamic. A static data structure has a fixed size. This meaning is different from the meaning of static modifier. Arrays are static; once we define the number of elements it can hold, the number doesn't change. A dynamic data

structure grows and shrinks at execution time as required by its contents. A dynamic data structure is implemented using links.

Data structures may further be categorized into linear data structures and non-linear data structures. In linear data structures every component has a unique predecessor and successor, except first and last elements, whereas in case of non-linear data structures, no such restriction is there as elements may be arranged in any desired fashion restricted by the way we use to represent such types.

Abstract Data Type Example

An Abstract Data Type consists of two things

- The custom VHDL data types and subtypes
- Operators that manipulate data of those custom types

Examples of ADTs include :

- Queue data type
- Finite state machine data type
- Floating and complex data type
- Vector and matrix data types

Notes:

Abstract data types (ADTs) are objects which can be used to represent an activity or component in behavioral modeling. An ADT supports data hiding, encapsulation, and parameterized reuse. As such they give VHDL some object-oriented capability.

An ADT is both a data structure (such as a stack, queue, tree, etc.) and a set of functions (e.g. operators) that provide useful services of the data. For example, a stack ADT would have functions for pushing an element onto the stack, retrieving an item from the stack, and perhaps several user-accessible attributes such as whether the stack is full or empty.

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 semantically 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.

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-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.

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.

And abstract data type is a generic defintion of the type of members and operators for the set. Such intergers, a stack, etc. It is not an actual instance, but the generic type that you could make instance of.

Defining a data type along with their operations are termed as Abstract Data Type. Some of the common ADT are Linked List, Stack, Queue, Trees. Consider Stack, its underlying data type may be array or linked list and opeartion's are pop(returns top most element & remove), push(add the given value into the top most).

- Abstract data types are data types which defines basic characteristic that is required for that data-types for example :-
    - Maps , which is used to create HashMaps
    - List , which is used to create ArrayList or other kind of list data-types. these are with respect to Java

It is an abstract definition or contract the API has to obey and promise the user to deliver what it says. There are two things in an API which can be considered pretty much independent of each other. The contract and the implementation, we mostly try to write a contract before providing an implementation.

This is good for two reasons, first – we define the goal and hence seal the requirement (with a minor scope for changes) before even starting the implementation, second – we make our intentions clear to the user of the API and give them a hint of what they can expect.

The ADT should be such that it can handle all sort of variations in the data structure, for

e.g. if I define an ADT for Tree, then it must be capable of handling Binary Tree, N-ary Tree, Red-Black Tree, AVL Tree, Heap etc.For example, integers are an ADT, defined as the values 0, 1, −1, 2, 2, ..., and by the operations of addition, subtraction, multiplication, and division, together with greater than, less than, etc. which are independent of how the integers are represented by the computer. Typically integers are represented in as binary numbers, most often as two's complement, but might be binary-coded decimal or in ones' complement, but the user is abstracted from the concrete choice of representation, and can simply use the data as integers. Another example can be a list, with components being the number of elements and the type of the elements. The operations allowed are inserting an element, deleting an element, checking if an element is present, printing the list etc. Internally, we may implement list using an array or a linked list, and hence the user is abstracted from the concrete implementation.