Data Structures and Algorithms for Engineers

Module 4: Abstract Data Types

Lecture 1: Abstract Data Types (ADT). Information hiding. Types and typing. Design Goals. Design practices.

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Abstract Data Types

- ADTs are an old concept
  - Specify the complete set of values which a variable of this **type** may assume
  - Specify completely the set of all possible operations which can be applied to values of this **type**
  - Do so without reference to the underlying implementation
    (hence abstract)
  - **Information hiding** (Dave Parnas)

Abstract Data Types

- It's worth noting that object-oriented programming gives us a way of combining (or **encapsulating**) both of these specifications in one logical definition
  - **Class** definition
    (data members and methods, i.e. function members)
  - **Objects** are instantiated classes
- Actually, object-oriented programming provides much more than this (e.g. inheritance and polymorphism)

Abstract Data Types

Typing and Data Types

- Data types allow programmers to specify what kind of data a variable (or data structure) can store

- Typing is necessary so a computer knows how values in memory will be represented
  - Native types typically include integer, floating point, character, string,…
  - Native data types are built into languages

- ADTs are programmer-defined data types that are created from native types or other ADTs with the express purpose of hiding certain complexities.

Abstract Data Types

An ADT …

- Hides the way information is stored and the details of how the operations do what they do
- Exposes “services” that programmers can use to access, add, delete, manipulate, and transform data
- Are designed for general use, without a particular application or program flow in mind

Abstract Data Types

Example

Native types

```
int myInteger;
char myLetter;
float aRealNumber;
```

Programmer defined type

```
#define SIZE 500
struct someStructType { /* stack is implemented as */
   char items[SIZE];    /* an array of items       */
   int num;             /* number of items         */
};

typedef struct someStructType myType; /* struct type */

myType stack;   /* stack is a struct data structure */
```

Abstract Data Types

Example

Native types

```
int myInteger;
char myLetter;
float aRealNumber;
```

Programmer defined type

```
#define SIZE 500
struct someStructType { /* stack is implemented as */
   char items[SIZE];    /* an array of items       */
   int num;             /* number of items         */
};

typedef struct someStructType *MyType; /* pointer type */

myType stack;  /* variable is a pointer to a stack */
```

Abstract Data Types

ADT Design Goals

- The primary goal in designing an ADT is to hide complexity and implementation details

- Encapsulation is the principle of hiding the way that data is structured, the algorithms used, and providing access to the data and services by way of well-defined interfaces

- We can design systems as a collection of related and interacting capsules that hide various complexities within them

Abstract Data Types

ADT Design Goals

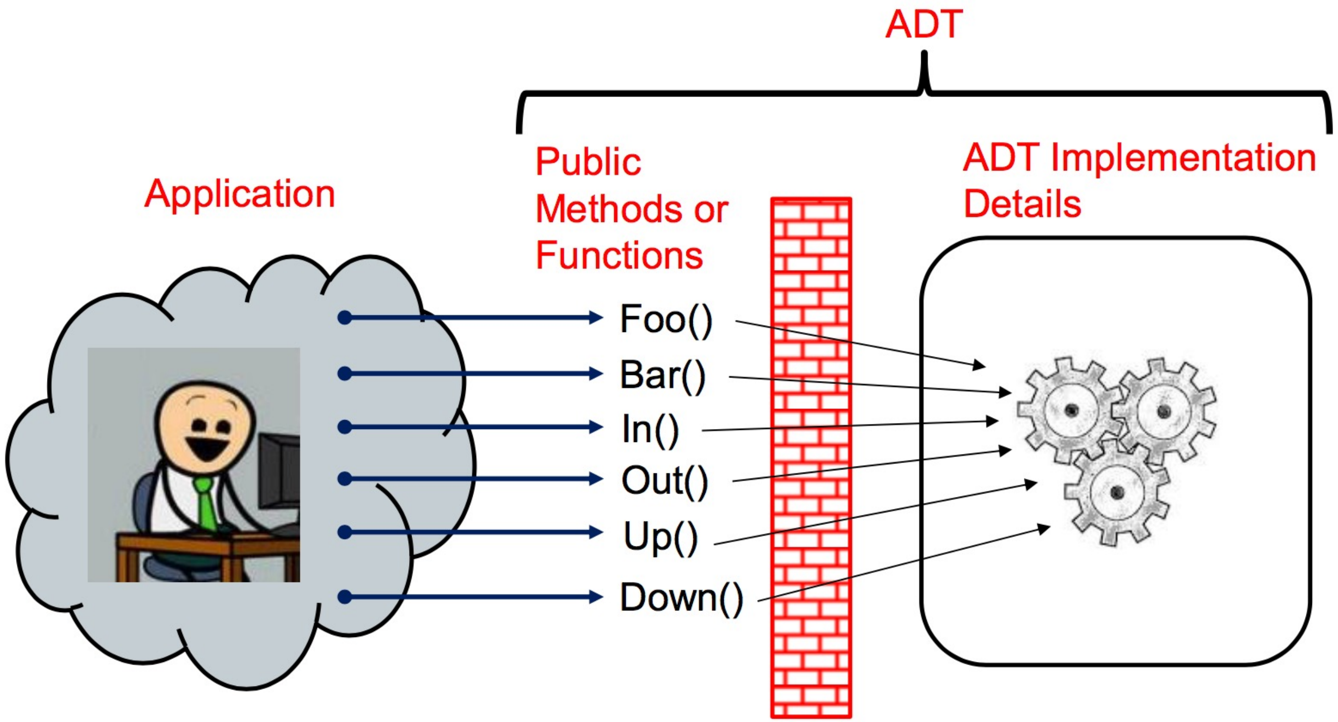
- We must decide
  - What data and operational details do we want to hide?
  - What we have to expose so that (application) programmers can do what they need to do
- We must design
  - The data organization (structure) … & decide how we will allocate resources, store and access data
  - The primitive data types or other ADTs that make up the ADT and the relationships between individual elements of the ADT we are creating
  - The internal and exposed (external) operations are required to operate on the data

Abstract Data Types

ADT Design Goals

- As in all software, in designing an ADT it must be correct:
  - Potentially many applications will depend upon the ADT
  - The data structure or algorithm should work correctly for all possible inputs that might be encountered

Abstract Data Types



Abstract Data Types

Pre- and post-condition checks should be built into ADTs

- protects callers from doing “bad things” and helps prevent logical defects during execution
- checks can be derived from assertions made during algorithm analysis
- The goal in assertion checking is to
  - check the input (preconditions)
  - check for termination (normal and abnormal)
  - check the result (post conditions)

Abstract Data Types

- Well designed and developed ADTs can
  - Improve usability of services
  - Hiding complexity, makes complex operations simple
  - Take the underlying implementation understandable
  - Facilitate reuse
  - Ease maintenance and modifiability
- Poorly designed ADTs can totally undermine these characteristics

Abstract Data Types

Design the ADT before you code

- Decide what the data and operations of the ADT will be before you write the code
- Consider modeling the ADT formally in terms of pre-conditions, post-conditions, invariants…
- ADT operations should do only 1 thing
- Reuse your own operations – never duplicate data or operations
- Think first. Code second.

Abstract Data Types

Decide what you will hide

- Hide as much as possible from the user of the ADT
- Create the most intuitive interfaces possible for programmers
- Reduce inter-module dependencies to the greatest extent possible

Abstract Data Types

- Comment your code – Code is written once and read many times
- Standardize to promote consistency
  - Use coding/comment standards and naming conventions
    use them on internal and external operations and data
- Include headers explaining what the code does
  - Traditionally called "headers" because they are comments at the beginning or "head" of the code body
  - Provides an overview of what the code module does and something about its history

Abstract Data Types

```
/********************************************************************************
*  Source File: FooBar.c
*  Description: This file contains routines for implementing foobar functions.
*  Author: Gill Bates
*  Initial Production Date: 5/5/07
*  Version: V1.2
*  Calling Convention: FooBar( int FooInput, int BarOutput );
*  Parameter List:
*  int FooInput – integer foo like data
*  int BarOutput – integer bar like data
*  Preconditions: FooInput must be greater than or equal to zero
*  Postconditions: BarOutput will be set to the relevant bar value based on FooInput
*  Functional Abstract: ...
*  *********************************************
*  Revision History
*  12/25/07: Changed FooInput from float to int. Author: Jack Sommers
*  07/04/08: Fixed problem with error message. Author: Jill Smith
```

Abstract Data Types

- Keep your code modules as simple as possible
  - many small, simple operations are often easier to understand than a single complex operation
- Don't be cute and clever with code
  - collapsing 5 lines of code into 1, may yield no advantage at all at execution time
  - complex code is defect prone
  - hard to understand what that 1 line of code does
  - it is better to be sure and correct, than cute and clever and maybe wrong!

Abstract Data Types

- Design error handling from the very beginning

- Good error handling is a result of good correctness and behavioral analysis:

  – error anticipation
  – how the application will react to various types of errors, especially if aborts/halts are possible
  – managing the consequences of errors