CSI33 Data Structures

Department of Mathematics and Computer Science
Bronx Community College

August 30, 2017

Outline

Outline

From Data Type to ADT

Values

A **value** is a unit of information used in a program. It can be associated with a constant or variable (a name) by an assignment statement:

```
person = 'George'
n = 4
```

All primitive type values (integer, float, string) are represented internally in the computer program's memory as a series of zeros and ones.

More complex types have values which are combinations of more primitive types.

From Data Type to ADT

Data Types

A **Data Type** is the set of possible values which all represent the same type of information and share the same behavior. In Python, most data types are classes, and a value of some data type is an object in that class.

```
int
str
float
list
dict
file
```

Defining an ADT = Specification

Data Abstraction

The data is represented using **abstract attributes**. Example: a `card` has attributes `suit` and `rank`. The behavior is given by specifying functions, with the signature , preconditions, and postconditions of procedural abstraction. **Data Abstraction** is the hiding of the primitive components comprising the values of some type, and hiding the implementation of the operations using that type.

Implementation of an ADT

Concrete Representation

The abtract attributes are represented using types from the programming language or previously defined classes. Example: a `suit` is now a member of the Python string class str, and is allowed to have the values `'c'`,`'d'`, `'h'` or `'s'`.

Class Specification

Python Classes

ADTs become **Python classes**, and their behaviors become **methods** for those classes.

Class Specification

Data Abstraction

In the class definition, a comment will tell how the concrete representation corresponds to the abstract attributes (for example, the letter `'c'` corresponds to the suit `'clubs'`).

Class Specification

Functional Abstraction

Each method specification includes a comment listing all preconditions and postconditions.

Class Implementation

Data Representation

In the class definition, the constructor `__init__` will take parameters to set the attributes of new objects (or use default values).

Class Implementation

Method Implementation

The actual Python code to implement the behavior of each method follows the comments listing preconditions and postconditions. Typical methods are **mutators**, which change attribute values, and **accessors** which return attribute values without changing them.

Changing the Representation

The Abstraction Barrier

By keeping the specification and implementation separate, it is possible to change the implementation—say, to use a more efficient algorithm—without having to change any program that uses the ADT respectful of its specification.

Changing the Representation

The Abstraction Barrier

The program which uses the ADT only has access to it through its methods, which are written to obey the specification—what types of parameters are passed and what types of values are returned. The concrete representation can change, but as long as the methods have the same signatures, and honors the same contracts (preconditions and postconditions), the program which calls them will still work.

Object Oriented Design (OOD) and Object Oriented Programming (OOP)

OOD extends ADTs

Data Abstraction is only one of several ideas that have helped to advance software engineering. Object Oriented Design and programming uses ADTs as well as other principles.

Object Oriented Design (OOD) and Object Oriented Programming (OOP)

Encapsulation

Also known as information hiding, this separates the issues of what to do from issues of how to do it.

Object Oriented Design (OOD) and Object Oriented Programming (OOP)

Polymorphism

This is the principle that sending the same message (that is, calling the same method) to objects in different classes should make the objects behave the same.

Object Oriented Design (OOD) and Object Oriented Programming (OOP)

Inheritance

Classes which share behaviors should not have to reimplement these behavior if they can be **inherited** from a base class which implements that same behavior. This principle promotes reuse of code, which in turn makes software more reliable, since bugs are more localized.