

Abstract Data Types

Contents

1. Description and and definition on an ADT
2. Terminology
3. Notation
4. Example – ADT WaterTank
5. Implementing an ADT description
6. Example – ADT Bag
7. Interface Bag

Abstract Data Types

- What does ‘abstract’ mean?
- From Latin: to ‘pull out’ - the essentials
 - To defer or hide the details
 - Abstraction emphasizes essentials and defers the details, making engineering artifacts easier to use
- I don’t need a mechanics understanding of what is under a car’s hood in order to drive it
 - *What’s the car’s interface?*
 - *What’s the implementation?*

Floating point numbers

- You don't need to know much about how floating point arithmetic works in order to use `float`
 - Indeed, the details can vary depending on processor, even virtual coprocessor
 - But the compiler hides all the details from you
 - some numeric ADTs are built-in
 - All you need to know is the syntax and meaning of operators, such as `+`, `-`, `*`, `/`, etc.
- Hiding the details of implementation is called **encapsulation (data hiding)**

ADT = properties + operations

- An **ADT** (Abstract Data Type) describes a set of objects sharing the same properties and behaviors
 - The **properties** of an ADT are its **data** (representing the internal state of each object)
 - `double d;` -- bits representing exponent & mantissa are its data or state
 - The **behaviors** of an ADT are its **operations** or **functions** (operations on each instance)
 - `d / 2;` //operators & functions are its behaviors
- Thus, an ADT couples its *data* and *operations*
 - OOP (object oriented programming) emphasizes **data abstraction**

Formal, language-independent ADTs

- An ADT is a formal description, not code; independent of any programming language
 - *Why is code independence a good idea?*
- Promotes **design by contract**:
 - Specify responsibilities of suppliers and clients explicitly, so they can be enforced, if necessary.

Abstract Data Types

An Abstract Data Type (ADT) is a description of the operations performed on a (collection of) data. It has the following components:

- **Name of the ADT**

- **Types represented in the collection of data**

Each type has previously been defined in an ADT (or is a well-known primitive type)

- **Functions that operate on the data**

Functions will specify the domain (parameters) and range (return type)

- **Preconditions for any function that is not a total function**

- **Post-conditions for each of the functions**

Abstract Data Types

Terminology

Total Function – A Total Function is one that is defined for every value in the domain. Correspondingly,

Partial Function – is a function that is defined only on a subset of the domain

Abstract Data Types

Examples

SQR (square) is a function that is defined for all real numbers

SQR: Real \rightarrow Real

SQRT (square root) is a partial function on the real numbers

SQRT: Real \rightarrow Real

Partial functions have a precondition specifying the subset of the domain over which they are defined.

The precondition for SQRT is that it is defined only for real numbers greater than or equal to 0.

Abstract Data Types

Terminology

For container classes transformations such as **remove** are usually partial functions, and observations such as **isEmpty** are almost always total functions.

Given the following function:

remove: Container, Integer -/-> Container

This partial function expresses the fact that remove takes as its arguments a Container object and an integer (index) and returns a transformed container. It is a partial function because it is not defined on the empty container and is only defined over integers greater than 0 and less than the size (number of items contained) of the container

Note! In this terminology a Container is mapped into another container much as SQR maps a real into a different real. The transformed container has a different state (expressed by its capacity and contents) and is therefore regarded as a different container.

Abstract Data Types

Notation

The list of function takes the following form:

Function Name :	Domain of definition (arguments)		Range (return type)
SQR	Real	\rightarrow	Real
SQRT	Real	$-/->$	Real
remove	Container , Integer	$-/->$	Container

The symbol \rightarrow denotes that the mapping is that of a total function, whereas

The symbol $-/->$ denotes a partial function

Abstract Data Types

Notation

Preconditions – all partial functions have a precondition that specifies the sub-domain over which they are defined

We express the precondition that SQRT is defined for real numbers greater than or equal to 0 as

For all $x \in \text{Real}$

$\text{pre-SQRT}(x) ::= x \geq 0$

$\text{pre-SQRT}(x) ::=$ translates as-- a precondition for SQRT acting on x , where x is any real number, is defined to be ($::=$) that x is greater than or equal to 0

Abstract Data Types

Similarly, the precondition for remove is written

For all $c \in \text{Container}$ and $i \in \text{Integer}$

$\text{pre-remove}(c, i) ::= \text{!empty}(c) \ \&\& \ 0 \leq i < \text{size}(c)$

empty and *size* are other functions in the ADT expressing observations on c

Abstract Data Types

Notation

Post-conditions describe the state of the collection after a function has acted.

We use the following notation to express post-conditions:

For all $x, y \in \text{Real}$

$\text{post-SQR}(x; y) ::= y = x * x \ \&\& \ y \geq 0$

$\text{post-SQRT}(x; y) ::=$ translates as – The post condition for the function SQR that maps a real number x into a real number y is defined to be y is equal to x times x and y is greater than or equal to 0.



Abstract Data Types

Notation

For the remove function, we can express the post-condition as follows:

For all $c, c' \in \text{Container}$, and all $i \in \text{Integer}$

$\text{post-remove}(c, i; c') :: = \text{size}(c') = \text{size}(c) - i$

arguments  **returned value** 

Abstract Data Types

There are four kinds of functions:

- 1. Create functions – implemented by constructors in programming languages**
- 2. Dissolve functions – implemented by garbage collector in runtime environment**
- 3. Transformations – change the state of one or more of the arguments**
- 4. Observations – return information about the arguments, but do not change the state of the collection**

For observations, the state of the system is the same after as it was before – post conditions only state the obvious (that a particular type of object has been returned) and therefore can be omitted from the ADT description

Abstract Data Types

Putting it all Together

Consider an ADT description of a Water Tank where we will use type Real to denote the amount of water held in the tank

ADT WaterTank

TYPES: Real, Boolean

FUNCTIONS

Create:	Real	-/->	WaterTank
Dissolve:	WaterTank	→	()
add:	WaterTank, Real	-/->	WaterTank
remove:	WaterTank, Real	-/->	WaterTank
isEmpty:	WaterTank	→	Boolean
contents:	WaterTank	→	Real
capacity :	WaterTank	→	Real

Abstract Data Types

ADT WaterTank (continued)

PRECONDITIONS

for all $t \in \text{WaterTank}$, $n, x \in \text{Real}$

pre-Create(n) ::= $n > 0$ //capacity of new tank > 0

//can't add more than tank can hold

pre-add(t, x) ::= $x > 0 \ \&\& \ x \leq \text{capacity}(t) - \text{contents}(t)$

//can't remove more than you have

pre-remove(t, x) ::= $x > 0 \ \&\& \ x \leq \text{contents}(t)$

Abstract Data Types

ADT WaterTank (continued)

POST-CONDITIONS $\leq \in \geq$

for all $t, t' \in \text{WaterTank}, x, y, n \in \text{Real}$

post-Create($n; t'$) ::= capacity(t') = n && isEmpty(t')

post-add($t, x; t'$) ::= contents(t') = contents(t) + x

post-remove($t, x; t'$) ::= contents(t') = contents(t) - x

New tank t' has a capacity of n and is empty



Abstract Data Types

Converting the ADT to a Class Description

The principal difference between the ADT representation of a WaterTank and a java class description is that

in the ADT the WaterTank is just one of the types in the collection and it can appear as an argument and/or as a return type for a function.

In a java class description, the WaterTank object is not an argument or a return type, but the recipient of a message that causes it to execute a method (function) that may change its state.

In **class** WaterTank, the function *Create* is replaced by a constructor, and the function *Dissolve* is performed automatically.

Abstract Data Types

Class WaterTank

```
public class WaterTank {  
    private double content, capacity; //data attributes  
    public WaterTank(double cap) throws TankNotBuiltException{  
        //precondition: cap > 0  
        //post-condition: new tank has capacity cap and content 0  
        if (cap > 0 ) { capacity = cap; content = 0; }  
        else  
            throw new TankNotBuiltException( );  
    }  
}
```

Abstract Data Types

Class WaterTank

```
public void add(double amt) throws TankAddException {  
    //precondition:    amt <= capacity – content  
    //post-condition: content = (old) content + amt  
    if (amt > 0 && amt <= capacity – content)  
        content = content + amt;  
    else  
        throw new TankAddException( );  
}
```

Abstract Data Types

class WaterTank (cont.)

public void remove(**double** amt) **throws** TankRemoveException {

//precondition: **amt <= content**

//post-condition: **content = (old) content – amt**

if (amt > 0 && amt <= content)

 content = content - amt;

else

throw new TankRemoveException();

}

//observations

public boolean isEmpty () { **return** content < 0.001; }

public double getContents () { **return** content; }

public double getCapacity () { **return** capacity; }

Abstract Data Types

Second Example – ADT Bag

A Bag is a very rudimentary container. You are able to put an Object into it, poke your hand in and take something out “without looking”, and “shake it” to see if it is empty. You will also be able to tell when the bag is full.

Let us write an ADT description of such a Bag

Abstract Data Types

ADT Bag

TYPES: **Object, Boolean, Integer**

FUNCTIONS

Create:	Integer	-/->	Bag
Dissolve:	Bag	→	()
put:	Bag, Object	-/->	Bag
grab:	Bag	-/->	Bag, Object
isEmpty:	Bag	→	Boolean
isFull:	Bag	→	Boolean

Abstract Data Types

ADT Bag

PRECONDITIONS

for all $b \in \text{Bag}$, for all $x \in \text{Object}$, for all $n \in \text{Integer}$

$\text{pre-Create}(n) ::= n > 0$

$\text{pre-put}(b, x) ::= \text{!isFull}(b)$

$\text{pre-grab}(b) ::= \text{!isEmpty}(b)$

Abstract Data Types

ADT Bag (cont.)

POST-CONDITIONS

for all $b, b' \in \text{Bag}$, for all $x \in \text{Object}$, $n \in \text{Integer}$

$\text{post-Create}(n; b') ::= \text{capacity}(b') = n \ \&\& \ \text{size}(b') = 0$

$\text{post-put}(b, x; b') ::= \text{!isEmpty}(b') \ \&\& \ \text{size}(b') = \text{size}(b) + 1$

$\text{post-grab}(b; b', y) ::= \text{!isFull}(b') \ \&\& \ \text{size}(b') = \text{size}(b) - 1$

Here we have used two functions **size** and **capacity** that are not functions defined in the ADT but whose meanings are obvious and needed to express the changes in the Bag that occur due to the two transformations.

Next – convert the ADT representation into a class definition. In this example, we express the ADT as an interface. It will be left to you as an exercise to implement this interface.

Abstract Data Types

Class Bag

```
public class Bag {  
    //in your implementtion you will need attributes size, capacity;  
    public Bag(int n);  
        //precondition:  n > 0  
        //post-condition: Bag of capacity n is created, size = 0  
    public boolean put(Object x);  
        //precondition:  Bag is not full  
        //post-condition: Size of the bag is increased by 1, return true if successful  
    public Object grab( );  
        //precondition:  Bag is not empty  
        //post-condition: Some Object is removed (and returned),  
        //                size is decremented by 1 – throws exception if empty  
    public boolean isEmpty( );  
    public boolean isFull( );  
}
```

Abstract Data Types

1. **Constructors:** these operations are used whenever we want to create a new object of that particular type. Constructors are required as the creation of a new object belonging to an ADT that requires a sequence of steps
2. **Destructors:** these are operations that are going to be executed when an object is removed from the system (because it is not used any longer)
3. **Inspectors:** these are operations that allows one to inspect the content of an object or test properties of the object (without modifying it)
4. **Modifiers:** these are operations that either modify an object (**Mutators**) or generate new objects (**Producers**).

Abstract Data Types

int is Java's primitive integer type.

int is immutable, so it has no mutators.

- **creators**: the numeric literals 0, 1, 2, ...
- **producers**: arithmetic operators +, -, \times , \div
- **observers**: comparison operators ==, !=, <, >
- **mutators**: none (it's immutable)

Abstract Data Types

List is Java's list interface. List is mutable.

List is an interface, which means that other classes provide the actual implementation of the data type. These classes include ArrayList and LinkedList.

- **creators:** ArrayList and LinkedList constructors, Collections.singletonList
- **producers:** Collections.unmodifiableList
- **observers:** size, get
- **mutators:** add, remove, addAll, Collections.sort

Abstract Data Types

String is Java's string type.

String is immutable.

- **creators:** String constructors
- **producers:** concat, substring, toUpperCase
- **observers:** length, charAt
- **mutators:** none (it's immutable)