Exercise 2: Abstract data types, Objects

Hand-out: 16 April 2004
Due: 23 April 2004

Master Solution

1. Summary: Abstract Data Types (ADTs)
An ADT specification is a formal, mathematical description that consists of four parts:
- TYPES: List of the (possibly generic) types introduced in the specification.
- FUNCTIONS: List of operations applicable to instances of the ADT.
- AXIOMS: List of properties of the values of the functions.
- PRECONDITIONS: In case of partial functions, specification of the domain.

Let’s take the example that was used in the lectures: the ADT of a stack. (Remember: A
stack object serves to pile up and retrieve other objects in a last-in, first out (“LIFO”) manner, the latest inserted element being the first one to be retrieved.)

- TYPES:
  \[ STACK [G] \]

- FUNCTIONS:
  \[ put: STACK [G] \times G \rightarrow STACK [G] \]
  \[ remove: STACK [G] \not\rightarrow STACK [G] \]
  \[ item: STACK [G] \not\rightarrow G \]
  \[ empty: STACK [G] \rightarrow BOOLEAN \]
  \[ new: STACK [G] \]

- PRECONDITIONS:
  \[ remove (s: STACK [G]) \text{ require not empty (s)} \]
  \[ item (s: STACK [G]) \text{ require not empty (s)} \]

- AXIOMS:
  For all \( x \): \( G \), \( s \): \( STACK [G] \)
  \[ item (put (s, x)) = x \]
  \[ remove (put (s, x)) = s \]
  \[ empty (new) \] (or: \( empty (new) = True \))
  \[ not empty (put (s, x)) \] (or: \( empty (put (s, x)) = False \))
2. Bank accounts

To do
Write an ADT specification for a “bank account” type with operations such as “deposit”, “withdraw”, “current balance”, “holder”, “change holder” and a lower “balance limitation”.

Hint
Think of how you would write a class BANK_ACCOUNT and how you would specify its features.

To hand in
Hand in the ADT of the “bank account” type.

Solution
• Hand in the ADT of the “bank account” type.

• TYPES:
  BANK_ACCOUNT

FUNCTIONS:
  new: BANK_ACCOUNT
  deposit: BANK_ACCOUNT × INTEGER ↦ BANK_ACCOUNT
  withdraw: BANK_ACCOUNT × INTEGER ↦ BANK_ACCOUNT
  current_balance: BANK_ACCOUNT → INTEGER
  Minimum_balance: BANK_ACCOUNT → INTEGER
  holder: BANK_ACCOUNT → STRING
  change_holder: BANK_ACCOUNT × STRING ↦ BANK_ACCOUNT

• PRECONDITIONS:
  deposit (a: BANK_ACCOUNT; i: INTEGER) require i >= 0
  withdraw (a: BANK_ACCOUNT; i: INTEGER) require i >= 0
     and current_balance (a) – i ≥ Minimum_balance (a)
  change_holder (a: BANK_ACCOUNT; s: STRING) require
     (not (s = holder (a)) and not (is_empty (s)))

• AXIOMS:
  For all a: BANK_ACCOUNT, i: INTEGER, s: STRING
  current_balance (a) ≥ Minimum_balance (a)
  current_balance (withdraw (deposit (a, i), i)) = current_balance (a)
  current_balance (deposit (a, i)) = current_balance (a) + i
  holder (change_holder (a, s)) = s
  current_balance (new) = Minimum_balance
3. Classes vs. Objects

It is important to make a clear distinction between classes and objects. A class is the representation of an abstract data type; it is static. An object is an instance of a class; it is dynamic (exists only at run time).

In *Object-Oriented Software Construction*, 2nd edition (p 167), Bertrand Meyer refers to a textbook extract that messes up between classes and objects. Here is the extract:

We might identify a “User” Object in a problem space where the system does not need to keep any information about the user. In this case, the system does not need the usual identification number, name, access privilege, and the like. However, the system does need to monitor the user, responding to requests and providing timely information. And so, because of required Services on behalf of the real world thing (in this case, User), we need to add a corresponding Object to the model of the problem space.

We need your help to get this text right.

**To do**
For each use of the word “object”, “thing” or “user” in that extract, underline the word in blue if you think that the authors really meant “object”; underline the word in red if you think that they really meant “class”.

**To hand in**
Hand in the extract with colors.

**Solution**
- For each use of the word “object”, “thing” or “user” in that extract, underline the word in blue if you think that the authors really meant “object”; underline the word in red if you think that they really meant “class”.

We might identify a **“User” Object** in a problem space where the system does not need to keep any information about the **user**. In this case, the system does not need the usual identification number, name, access privilege, and the like. However, the system does need to monitor the **user**, responding to requests and providing timely information. And so, because of required Services on behalf of the real world **thing** (in this case, **User**), we need to add a corresponding **Object** to the model of the problem space.
4. Try your hand with EiffelStudio

Now, it’s time to play with EiffelStudio and apply what you’ve learnt during the first two exercise sessions.

To do

- Launch EiffelStudio.
- Select “Create a new project”:

![Create a new project dialog]

- Click “Next”. The following dialog appears:

![Choose your project name and directory]

Replace the system name “sample” by “bank_account”.
Leave the root cluster name “root_cluster” unchanged.
Replace the root class name “ROOT_CLASS” by “BANK_ACCOUNT”.
Leave the root feature name “make” unchanged.
Select the location where you want the code to be generated.
Click “OK”. EiffelStudio will start compiling.
• Write the class name \textit{BANK	extunderscore ACCOUNT} in the corresponding text field (top left of EiffelStudio):

The generated class text appears in the editor:

- 

• Complete the class \textit{BANK	extunderscore ACCOUNT} according to the ADT specification you wrote in part 2.
• Compile your project by clicking the “Compile” button:

• Go back to the code if it does not compile.

\textbf{Hint}

Use the keyword \texttt{require} for preconditions and the keyword \texttt{invariant} for axioms. (See document “\emph{Eiffel: The Essential}” given during the first exercise session.)

\textbf{To hand in}

Hand in the text of class \textit{BANK	extunderscore ACCOUNT}.

Submit paper solution to your assistant (Make sure your project compiles. If it does not, explain your problems to your assistant.)
Solution
- Complete the class `BANK_ACCOUNT` according to the ADT specification you wrote in part 2.

```plaintext
indexing

description: "Representation of a bank account"

class

BANK_ACCOUNT

create

make

feature -- Initialization

make is

-- Initialize the account with `Minimum_balance'.
do
current_balance := Minimum_balance
ensure
minimum_balance: current_balance = Minimum_balance
end

feature -- Access

current_balance: INTEGER
-- Current balance of the account

Minimum_balance: INTEGER is 1000
-- Minimum amount of money on the account

holder: STRING
-- Name of the person who holds the account

feature -- Basic operations

deposit (an_amount: INTEGER) is
-- Deposit `an_amount' on the account.
require
an_amount_positive: an_amount >= 0
do
current_balance := current_balance + an_amount
ensure
increased:
    current_balance = old current_balance + an_amount
end
```
withdraw (an_amount: INTEGER) is
   -- Withdraw `an_amount' from the account.
   require
      an_amount_positive: an_amount >= 0
      an_amount_not_too_big:
         current_balance - an_amount >= Minimum_balance
do
   current_balance := current_balance - an_amount
ensure
   withdrawn:
      current_balance = old current_balance - an_amount
end

feature -- Element change

change_holder (a_new_holder: STRING) is
   -- Change current `holder' of the bank account
   -- to `a_new_holder'.
   require
      a_new_holder_not_void: a_new_holder /= Void
      a_new_holder_not_empty: not a_new_holder.is_empty
      is_new_holder: a_new_holder /= holder
do
   holder := a_new_holder
ensure
   holder_changed: holder = a_new_holder
end

invariant
   consistent: current_balance >= Minimum_balance
end