Exercise 2: Abstract data types, Objects

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

Please solve this exercise alone.

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**:
  \[ \text{STACK} [G] \]

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

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

- **AXIOMS**:
  
  For all \( x: G, s: \text{STACK} [G] \)
  
  \[ \text{item} (\text{put} (s, x)) = x \]
  
  \[ \text{remove} (\text{put} (s, x)) = s \]
  
  \[ \text{empty} (\text{new}) \]
  
  \[ \text{not empty} (\text{put} (s, x)) \]
  
  \[ (\text{or: empty} (\text{new}) = \text{True}) \]
  
  \[ (\text{or: empty} (\text{put} (s, x)) = \text{False}) \]
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.

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, 2\textsuperscript{nd} 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.
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”:

![EiffelStudio dialog](image)

- Click “Next”. The following dialog appears:

![Choose your project name and directory](image)

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 *BANK_ACCOUNT* in the corresponding text field (top left of EiffelStudio):

```plaintext
Class BANK_ACCOUNT
```

The generated class text appears in the editor:

```plaintext
indexing
  description : "System's root class"
  note       : "Initial version automatically generated"

class
  BANK_ACCOUNT

create
  make

feature -- Initialization
  make is
    -- Creation procedure.
    do |
    --| Add your code here
  end
end -- class BANK_ACCOUNT
```

• Complete the class *BANK_ACCOUNT* according to the ADT specification you wrote in part 2.

• Compile your project by clicking the “Compile” button:

```
Class BANK_ACCOUNT

Compile

Features +
```

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

**Hint**

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

**To hand in**

Hand in the text of class *BANK_ACCOUNT*.

Submit paper solution to your assistant (Make sure your project compiles. If it does not, explain your problems to your assistant.)