Classroom exercise 1

7 May 2004

Master Solution

1. Objects (18 points)

Feature categories

• Give two classifications of features. You may want to use a picture. (8 points)
• Give the two O-O principles corresponding to these two classifications. (4 points)

- Feature classification by role (Command / query separation)
  - A **command** returns no result; a **query** returns a result.
  - If a query is computed at run time, it is called **function**; if it is stored in memory, it is called an **attribute**.

- Feature classification by implementation (Routine / attribute separation)
  - A **routine** is a computation (algorithm) applicable on instances of the class; an **attribute** is stored in memory.
  - If a routine returns a result, it is called a **procedure**; otherwise, it is called **function**.
**Garbage collection**

- What does it mean for a Garbage Collector (GC) to be consistent? (2 points)
  
The GC never reclaims used space.

- What does it mean for a GC to be complete? (2 points)
  
The GC always reclaims unused space (eventually).

- Explain the pros and cons of automatic garbage collection. (4 points)

  **Pros:**
  - Manual reclamation is dangerous. Hampers software reliability.
  - In practice, bugs arising from manual reclamation are among the most difficult to detect and correct. Manifestation of bug may be far from source.
  - Manual reclamation is tedious: need to write “recursive dispose” procedures.
  - Modern garbage collectors have acceptable overhead (a few percent) and can be made compatible with real-time requirement.
  - GC is tunable: disabling, activation, parameterization, …

  **Cons:**
  Performance concerns (although usually not justified, except in real-time systems).

**2. Genericity (6 points)**

**Class vs. Type**

- Give the definition of a type. (2 points)
A type is the description of a set of run time objects. Every type is based on a class called its base class.

- Is there a one to one correspondence between a type and a class? (2 points)

In case of non-generic classes, a class corresponds to one and only one type. However, in case of generic classes, there is no one to one correspondence between a type and a class.

- Explain the difference between a class and a type. (2 points)

In the case of non-generic classes, classes and types are the same; they describe the general notion of a “module”.

The only difference is in the case of genericity: A generic class \( C[G] \) is like a “type pattern”; it covers an infinite set of possible types. One of these types can be obtained by providing an actual generic parameter – itself a type – corresponding to \( G \).

3. Design by Contract (60 points)

Different kinds of contracts

- List the different kinds of assertions, and explain the role of each. (12 points)

Contracts — also called assertions — are boolean expressions stating the semantic properties of classes. They are used to express a specification. For instance, a class \( LIST \) may have an assertion stating that the number of elements in the list must be smaller than or equal to the list’s capacity. Such contracts between software elements include:

- **Preconditions**: Requirements under which routines will function properly; they are conditions binding on the client (the caller), turning into benefits for the supplier.

- **Postconditions**: Properties guaranteed on routine exit; they are conditions incumbent on the supplier on completion of the task, turning into benefits for the client.

- **Class invariants**: Fundamental semantic constraints applying to a class; they express properties, which are still valid after each routine call.

- **Check instructions**: Expressions ensuring that a certain property is satisfied at a specific point of a method’s execution; they help document a piece of software and make it more readable for future implementers.

- **Loop invariants**: Conditions which have to be satisfied at each loop iteration and when exiting the loop; they aim at guaranteeing a loop is correct.
Loop variants: Integer expressions ensuring that a loop is finite.

Benefits

- Describe four key benefits of Design by Contract. Explain. (8 points)

Benefits of contracts are too often restricted to the debugging and testing aspects. Yet the use of contracts has many other beneficial consequences on the software process. The four main applications are:

- **Software correctness**: Contracts help having software right in the first place, as opposed to the more common approach of trying to debug software into correctness. This first use of contracts is purely methodological: the Design by Contract method advises you to think about each routine’s requirements and write them as part of your software text; think and write the global properties of your class, find out the properties of loops, and so on. This is only a method, some guidelines for software developers, but it is also perhaps the main benefit of contracts, that is help you design and implement correct software right away.

- **Documentation**: Contracts enable you to have documentation about your software, through what is known in Eiffel as the class contract form. And having documentation is essential in the production of reusable software elements and more generally in organizing the interfaces of modules in large software systems. Preconditions, postconditions and class invariants provide potential clients of a module with basic information about the services offered by the module, which is expressed in a concise and precise form. The key point is that this is no verbose documentation but a set of carefully expressed expressions, which appear in the software itself.

- **Debugging and testing**: Contracts make it much easier to detect “bugs” in a piece of software, since the program execution just stops at the mistaken point. It becomes even more obvious with assertions tags, i.e. identifiers before the assertion text itself.

- **Management**: Contracts help understand the global purpose of a program without having to go into the depth of the code, which is especially appreciable when you need to explain your work to less-technical persons.

The principles of Design by Contract are especially crucial when reusing existing software: to be reusable, a component must be equipped with a clear specification of its working conditions through preconditions, postconditions and class invariants, which also have to be part of the software itself. The crash of the Ariane 5 launcher described in OOSC2 (page 410) is a well-known example, which is likely to convince the most reluctant C++, Java or C# developer of the usefulness of contracts!
**Class correctness**

- What does it mean for a class to be correct? (2 points)

Correctness is a relative notion. A class is correct if its implementation is consistent with its specification. (Of course, this assumes there is a specification!)

- Give a formal definition. (2 points)

For every creation procedure $cp$, the following property must hold:

$\{\text{Pre}_{cp}\} \text{ do } \{\text{Post}_{cp} \text{ and } \text{INV}\}$

meaning that any call to $cp$ starting in a state satisfying $cp$’s precondition $\text{Pre}_{cp}$ terminates in a state satisfying $cp$’s postcondition $\text{Post}_{cp}$ and the class invariant $\text{INV}$.

For every exported routine $r$, the following property must hold:

$\{\text{INV and Pre}_{r}\} \text{ do } \{\text{Post}_{r} \text{ and } \text{INV}\}$

meaning that any call to $r$ starting in a state satisfying $r$’s precondition $\text{Pre}_{r}$ and the class invariant $\text{INV}$ terminates in a state satisfying $r$’s postcondition $\text{Post}_{r}$ and the class invariant $\text{INV}$.

**Contract violation**

- What does a precondition violation express? (2 points)

A precondition violation expresses a bug in the client code.

- What does a postcondition violation express? (2 points)

A postcondition violation expresses a bug in the supplier code.

- What does a class invariant violation express? (2 points)

An invariant violation expresses a bug in the supplier code.

**Feature calls in assertions**

Consider a function `my_function` such as

```plaintext
my_function is
  -- Do something.
  require
    some_query
do
  -- Something here
ensure
  another_query
end
```

Explain the rules governing `some_query` and `another_query` to be able to use them in the routine contracts. (4 points)

Because `some_query` is involved in the precondition of `my_function`, clients of `my_function` must be able to check it. Therefore `some_query` must be exported at least as much as `my_function`. However, there is no restriction regarding the export status as `another_query`. Indeed, a routine postcondition may involve non exported routines to give implementation information to the developer, which clients do not need to know about.

**Introducing contracts**

Add contracts to the following class `ACCOUNT` and complete the comments if necessary. (16 points)

```plaintext
class ACCOUNT

create

feature {NONE} -- Initialization
    make (an_amount: like balance) is
    do
        balance := an_amount
    end

feature -- Access

    balance: INTEGER
        -- Account balance

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

feature -- Deposit

    deposit (an_amount: like balance) is
    do
        balance := balance + an_amount
    end

feature -- Withdrawal
```
withdraw (an_amount: like balance) is
        -- Subtract an_amount from current balance.
        do
            balance := balance - an_amount
        end

feature -- Status report

may_withdraw (an_amount: like balance): BOOLEAN is
        -- May an_amount be withdrawn from the account?
        do
            Result := (balance-an_amount >= minimum_balance)
        end

Here is the corresponding class with the contracts:

class
    ACCOUNT
create
make

feature {NONE} -- Initialization

make (an_amount: like balance) is
        -- Set balance to an_amount.
        require
            an_amount_is_big_enough:
                an_amount >= minimum_balance
        do
            balance := an_amount
        ensure
            balance_set: balance = an_amount
        end

feature -- Access

balance: INTEGER
        -- Account balance

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

    deposit (an_amount: like balance) is
    -- Add an_amount to current balance.
    require
        an_amount_is_non_negative: an_amount >= 0
    do
        balance := balance + an_amount
    ensure
        balance_increased: balance = old balance + an_amount
    end

feature -- Withdrawal

    withdraw (an_amount: like balance) is
    -- Substract an_amount from current balance.
    require
        may_withdraw: may_withdraw (an_amount)
    do
        balance := balance + an_amount
    ensure
        balance_decreased: balance = old balance - an_amount
    end

feature -- Status report

    may_withdraw (an_amount: like balance): BOOLEAN is
    -- May an_amount be withdrawn from the account?
    do
        Result := (balance-an_amount >= minimum_balance)
    ensure
        definition:
            Result = (balance-an_amount >= minimum_balance)
    end

invariant

    consistent: balance >= minimum_balance
end

Contract extraction

Here is an extract of the documentation provided with the .NET Framework for method Insert of class System.Collections.ArrayList:
public virtual void Insert (int index, Object value);

Inserts an element into the ArrayList at the specified index.

**Parameters**

index
The zero-based index at which `value` should be inserted.

value
The `Object` to insert.

**Exceptions**

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgumentOutOfRangeException</td>
<td>index is less than zero.</td>
</tr>
<tr>
<td></td>
<td>- or -</td>
</tr>
<tr>
<td></td>
<td>index is greater than Count.</td>
</tr>
<tr>
<td>NotSupportedException</td>
<td>The ArrayList is read-only.</td>
</tr>
<tr>
<td></td>
<td>- or -</td>
</tr>
<tr>
<td></td>
<td>The ArrayList has a fixed size.</td>
</tr>
</tbody>
</table>

**Remarks**

If `Count` already equals `Capacity`, the capacity of the list is doubled by automatically reallocating the internal array before the new element is inserted.

Write a complete specification of this feature (in Eiffel syntax) including preconditions and postconditions. (The preconditions and postconditions may involve calls to other class functions; if so, spell out the functions.) (10 points)

```eiffel
class
  ARRAY_LIST
feature -- Element change
  insert (an_index: INTEGER; a_value: ANY) is
    -- Insert `a_value` into the list at index `an_index`.
    require
    an_index_non_negative: an_index >= 0
    an_index_smaller_than_count: an_index <= count
    writable: not is_read_only
    extendible: not is_fixed_size
    do
      -- Something
```
4. Inheritance (14 points)

**Terminology**

Define the following terms:

- **Parent** (2 points)
  A class from which the given class inherits.

- **Child** (2 points)
  A class that inherits from the given class.
• Heir (1 point)
A class that inherits from the given class.

• Ancestor (2 points)
The class itself, or one of its direct or indirect parents.

• Proper ancestor (1 point)
A direct or indirect parent of the class.

• Descendant (2 points)
The class itself, or one of its direct or indirect heirs.

• Proper descendant (1 point)
A direct or indirect heir of the class.

• Instance (2 points)
An object built according to the mold defined by the class or any one of its proper descendants.

• Direct instance (1 point)
An object built according to the mold defined by the class.