1. Quality issues (5 points)

Question 1: Lifecycle (5 points)
1.1 Describe the Waterfall model of the software lifecycle. (Use a picture if needed.)
(2.5 points)

Validation means checking that the system meets its requirements (“doing the right things”); verification means checking the system “does things right”.

1.2 Explain the pros and cons of this model (2.5 points).
Pros:
- All activities described in the waterfall model are necessary
- The order is the right one.

Cons:
- Late appearance of actual code
- Lack of support for requirements change – and more generally for extendibility and reusability.
- Lack of support for the maintenance activity
- Division of labor hampering total quality
- Impedance mismatches
- Highly synchronous model

2. **Modularity, reusability (5 points)**

**Question 2: Modularity (2.5 points)**
Some programming languages require a main program to execute a piece of software. What are the advantages and disadvantages of this approach?

**Pros:**
A system using a “main” program is already more modular than a system where each part would communicate to each other, as illustrated on the next picture:

```
“Monarchy”
```
```
“Anarchy”
```

It follows the “few interface principle”.

**Cons:**
Because everything is centralized, an error in the main program results in a system that does not work. It’s not resistant to faults.
An object oriented approach will try to follow the following scheme (with no main program):

```
O-O
```

Question 3: Information hiding (2.5 points)

Define the Information hiding principle.

The underlying question is “how does one “advertise” the capabilities of a module”:
- Every module should be known to the outside world through an official, “public”
  interface.
- The rest of the module’s properties comprises its “secrets”.
- It should be impossible to access the secrets from the outside. (The clients may
  know about the secrets, but they should not be able to use them: they should not
  rely on it.)

The Information hiding principle says: “The designer of every module must select a
subset of the module’s properties as the official information about the module, to be
made available to authors of client modules.

3. Abstract Data Types (10 points)

Question 4: Naming style (4 points)

The class STACK of EiffelBase uses the feature names put, remove, and item to denote
the services that are usually called push (adding an element at the top of the stack), pop
(removing the top-most element of the stack) and top (returning the top-most element of
the stack). Explain why.

If one wants to have structure specific names, one would end up with names like:
- push, pop, and top for STACKs
- add, remove_oldest and oldest for QUEUES
- …

which means that developers need to learn a completely different terminology when
using one class or another.

The benefit of using put, remove, and item for all container classes is to have consistent
naming. Therefore, new users can learn more quickly: when exploring a new class
which follows the standard style, they understand it right away and can choose the
features they need much more easily.

Question 5: Mathematics vs. Software (4 points)

Explain the core difference between an ADT and a class.

A class is an implementation of an ADT. It is both a module and a type.

In the ADT “world”, one talks about mathematics, where there is no notion of change.
The functions described in an ADT, for example:

are side-effect free. In the above example, calling put will not change the original stack
to add a new element to it, but it will create a new stack that has one more element.

In the software world, there is a notion of procedure that can change objects; besides,
not all functions are pure functions, they may have side effects.
**Question 6: Sufficient completeness (2 points)**
Define the concept of sufficient completeness for an ADT.

A sufficiently complete specification is a “query expression” of the form \( f(\ldots) \) where \( f \) is a query, i.e.

\[
T \times \ldots \rightarrow \text{OTHER} \quad (\text{e.g. item, empty})
\]

may be reduced through application of the axioms to a form not involving \( T \).

**4. Objects (24 points)**

**Question 7: Return vs. Result (3 points)**
Eiffel uses the keyword *Result* to denote the result of a function, whereas other O-O languages like C# or Java use a *return* instruction.
Explain the differences between these two approaches.

*Result* is initialized to the default value of the returned result.

Having a *return* instruction instead has two main disadvantages:
- One needs an extra local variable to be able to compute the result piece by piece.
- One may need several return instructions.

**Question 8: Cloning (5 points)**
8.1 Define shallow cloning and deep cloning. (2 points)
8.2 Suppose you want to clone an instance of `LINKED_LIST`, what kind of cloning facility would you like to use? Explain why. (3 points)

8.1 Shallow cloning an object means creating a new object identical to the original. Deep cloning an object also creates a new object identical to the original, but it also duplicates all objects to which the original object was pointing recursively.
Shallow cloning is achieved in Eiffel by the feature *clone*; deep cloning by *deep_clone*.
8.2 To clone an instance of LINKED_LIST, one needs a cloning facility that is between shallow and deep cloning. Indeed, shallow cloning would just clone the list header, which is not enough (problem of dynamic aliasing). But deep cloning would be too much because it would also clone the objects in the list and the references to these objects, basically the whole program!

In Eiffel, it is possible to achieve the desired result by redefining copy (inherited from ANY in class LINKED_LIST). The feature clone (frozen in class ANY) will be redefined automatically.

Question 9: Export status of attributes in Eiffel (12 points)
Let’s consider a class DEVICE defined as follows:

```eiffel
class DEVICE
feature -- Access
    temperature: REAL
        -- Temperature captured by the device
end
```

9.1 Explain what isn’t valid in: (2 points)

```eiffel
class DEVICE_CLIENT
feature -- Basic operations
    print_temperature is
        -- Print temperature read on the device.
```

```eiffel
b := a
c := clone (a)
d := deep_clone (a)
```
local
  a_device: DEVICE
do
  create a_device
  print (a_device.temperature)
end

change_temperature (t: REAL) is
  -- Replace temperature read on device by t.
local
  a_device: DEVICE
do
  create a_device
  a_device.temperature := t
a_device: DEVICE
end

9.1 The direct assignment to the attribute temperature of a_device of class DEVICE:

\[ a\_device.\text{temperature} := t \]

is not allowed in Eiffel; it is syntactically illegal.

9.2 It’s forbidden to assign a value \( t \) to an expression \( a\_device.\text{temperature} \). In Eiffel, exported ("public") attributes are read-only. Why? Because allowing write access on attributes opens up all kinds of violations of the information hiding principle. It’s not extendible; it is tedious, unsafe, fragile. One does not have complete control on what the clients do. Besides, you may want to allow only certain temperature values and have a precondition, hence the need for a set procedure, rather than a direct assignment that you cannot control.

9.3 The Information Hiding principle.

9.4 To get write privileges, it’s required to write a “setter” procedure set_temperature. Hence the resulting class:

```eiffel
class DEVICE_CLIENT
feature -- Basic operations
  print_temperature is
    -- Print temperature read on the device.
    local
      a_device: DEVICE
    do
      create a_device
```
print (a_device.temperature)

change_temperature (t: REAL) is
  -- Replace temperature read on device by t.
  local
  a_device: DEVICE
  do
    create a_device
    a_device.set_temperature (t)
  end
end

with (in class DEVICE):
  set_temperature (t: REAL) is
    -- Set temperature to t.
    do
      temperature := t
    ensure
      temperature_set: temperature = t
    end

9.5 C# uses the notion of “property”, which is a pair of two features (a getter and a setter). For example:

```csharp
public int temperature
{
    get
    {
        return temperature_internal;
    }
    set
    {
        temperature_internal = value;
    }
}

private int temperature_internal;
```

**Question 10: Garbage collection (4 points)**

10.1 What does it mean for a Garbage Collector (GC) to be consistent? (1 point) To be complete? (1 point)

10.2 Explain the pros and cons of automatic garbage collection. (2 points)

Hint: Think about the problems of manual GC, which automatic GC addresses.

**10.1 Consistent:** the GC never reclaims used space.
**(Complete:** the GC always reclaims unused space (eventually).

**10.2 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).

6. Genericity (5 points)

Question 11: Constrained genericity (5 points)

11.1 What does the following notation mean? (2.5 points)

```
class C [G -> COMPARABLE]
```

11.2 Give an example using constrained genericity. (2.5 points)

11.1 It means that the actual generic parameter needs to conform to COMPARABLE.

11.2 Here is an example using constrained genericity:

```
class VECTOR [G -> NUMERIC]

feature
  infix "+" (other: VECTOR [G]): VECTOR [G] is
    -- Sum of current vector and other
    require
      other_not_void: other /= Void
      lower_is_valid: lower = other.lower
      upper_is_valid: upper = other.upper
    local
      a, b, c: G
    do
      create Result.make (lower, upper)
      from
        i := lower
      until
        i > upper
      loop
        a := item (i)
        b := other.item (i)
        c := a + b -- Requires a "+" operation on G
        Result.put (c, i)
        i := i + 1
      end
    end

end
```
7. Inheritance (5 points)

Question 12: Deferred classes (5 points)
12.1 What is the main difference between Java/C# interfaces and Eiffel deferred classes? (2.5 points)
12.2 May a deferred class have assertions? Justify why. (2.5 points)

12.1 Java/C# interfaces cannot contain any implementation at all: none of the features may be implemented; it cannot contain any attributes.
In Eiffel, a deferred class can contain attributes, and may be partially (or totally) implemented.

12.2 A deferred class can contain any kind of assertions (routine pre- and postconditions, class invariants, etc.). Indeed, it is just a “normal” class, which can be partially or even fully implemented; the only difference is that it cannot be instantiated.

8. Design by Contract (20 points)

Question 13: Assertions vs. assert (4 points)
Explain the differences between the assert instruction of C/C++ and assertions as defined in Design by Contract.

Assertions defined in Design by Contract go further than the assertion instruction of C/C++:
- Assert does not provide a contract. It does not bind a client and a supplier like preconditions, postconditions and class invariants do; it is just a test instruction.
- Clients cannot see asserts as part of the interface.
- Asserts do not have associated semantic specifications.
- Not explicit whether an assert represents a precondition, postcondition, or a class invariant.
- Asserts do not support inheritance. (In Design by Contract, there are precise rules applying to contracts when a feature gets redefined or a programmer defines a new descendant of a particular class.)
- Asserts do not yield automatic documentation (contrary to contracts that are the basis for generating software documentation automatically).

Question 14: Hoare triples (6 points)
14.1 Explain what the following notation means. (2 points)

\[ \{ P \} A \{ Q \} \]

14.2 What is the weakest possible precondition? The strongest? (2 points)
14.3 What is the weakest possible postcondition? The strongest? (2 points)

14.1 It means that any execution of A (where A is a set of instructions) started in a state satisfying P (the properties that must be true before execution, i.e. the preconditions) will terminate in a state satisfying Q (the properties of the state after execution, i.e. the postconditions).
14.2 The weakest possible precondition is True. The strongest possible precondition is False.

14.3 The weakest possible postcondition is True. The strongest possible postcondition is False.

**Question 15: Loop contracts (6 points)**

15.1 What is a loop variant? What is it used for? (2 points)
15.2 What is a loop invariant? What is it used for? (2 points)
15.3 Add the loop variant and invariant to the following example (2 points):

```pascal
maximum_of_array (an_array: ARRAY [INTEGER]): INTEGER is
  -- Maximum of array an_array
  require
  an_array_not_void: an_array /= Void
  an_array_not_empty: an_array.count > 0
  local
  i: INTEGER
  do
  from
  i := an_array.lower
  Result := an_array @ i
  until
  i = an_array.upper
  loop
  i := i + 1
  Result := Result.max (an_array @ i)
  end
  end
end
```

15.1 A loop variant is an integer expression used to ascertain termination of a loop. The loop’s initialization (from clause) sets the variant to a non-negative value. Any execution of the loop body decreases its value but maintains it non-negative.

15.2 A loop invariant is a boolean expression (an expression) used to deduce properties of the loop result. The loop’s initialization (from clause), when started in a state that does not satisfy the exit condition, preserves the loop invariant (in other words, it leaves the loop invariant true if it finds it originally true. As a result, the loop invariant will be satisfied on loop exit together with the exit condition.

15.3
from
    i := an_array.lower
Result := an_array @ i

invariant
    -- Result is the maximum of the elements
    -- of an_array at indices an_array.lower to i

variant
    an_array.upper - i + 1

until
    i = an_array.upper

loop
    i := i + 1
    Result := Result.max (an_array @ i)
end
end

Question 16: Imperative vs. Applicative (4 points)

Explain the difference(s) between

    do
        balance := balance - sum
    and
        ensure
            balance = old balance - sum

The following table summarizes the differences:

<table>
<thead>
<tr>
<th>do</th>
<th>ensure</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance := balance - sum</td>
<td>balance = old balance - sum</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>PRESCRIPTIVE</th>
<th>DESCRIPTIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>How?</td>
<td>Operational</td>
<td>Denotational</td>
</tr>
<tr>
<td></td>
<td>Implementation</td>
<td>Specification</td>
</tr>
<tr>
<td>Command</td>
<td>Query</td>
<td>Expression</td>
</tr>
<tr>
<td>Instruction</td>
<td>Applicative</td>
<td></td>
</tr>
</tbody>
</table>

Here are a few explanations:

- It is the difference between the prescriptive and the descriptive. More precisely: the instruction is prescriptive, it describes the “how” whereas the assertion is descriptive, it describes the “what”.
- The instruction is an operation whereas the assertion is a denotation, a meaning. It is an expression, not an instruction.
- It is the opposition between the implementation on the one hand and the specification on the other hand.
• Another point is that the instruction is a command: it does something, whereas the assertion is a boolean query, it asks for something, checks that a Boolean property is true.
• The last difference is the opposition between the imperative (the instruction) and the applicative (the assertion). Indeed, instructions are part of the “computer world”, they can change the state of the hardware/software machine, whereas assertions are mathematical objects, which never can change anything: they are side-effect free.

9. Persistence (5 points)

Question 17: Schema evolution (5 points)

Explain the problem of schema evolution. Sketch some approaches addressing the issue.

It’s possible that the class corresponding to a stored object changes. What to do at retrieval then?
The problem of schema evolution arises when at least one class used by the retrieving system retrieves a particular object whose own generating class was different in the storing system.

Different approaches to address the issue:
• Forsake previously stored objects.
• Offer a migration path from old format to new:
  o One-time conversion of old objects
  o Not applicable to a large persistent store or to one that must be available continuously
• Most general solution: on-the-fly conversion, including 3 steps:
  o Detection: catch object mismatches
  o Notification: make retrieving system aware of object mismatch
  o Correction: bring mismatched object to a consistent state and make it a correct instance of the new class version.

10. Design principles (6 points)

Question 18: Principles (6 points)

18.1 What is the Command-Query separation principle? (2 points)
18.2 What is the Option-Operand separation principle? Give an example. (2 points)
18.3 What are the advantages of a design applying the Option-Operand separation principle? (2 points)

18.1 A command (procedure) does something but does not return a result. A query (function and attribute) returns a result but does not change the state (“asking a question should not change the answer”); they should not produce abstract side effects.
18.2 The arguments of a routine should only include operands (no option). An argument is an option if, assuming the client had not supplied its value it would not have been possible to find a reasonable default. In the evolution of a class, arguments tend to remain the same, but options may be added and removed. In other words, operands are values on which a feature will operate; options are modes that govern how a feature will operate.

For example:
- Displaying a window:
  
  ```eiffel
  my_window.display (x_position, y_position, height, width, color, …)
  ```
  
  All arguments are options. It should be simply:
  
  ```eiffel
  my_window.display
  ```

- Printing a real:
  
  ```eiffel
  print (real_value, number_of_significant_digits, zone_length, …)
  ```
  
  The number is an operand; the other arguments are options. It should be:
  
  ```eiffel
  print (real_value)
  ```

18.3 It helps increase the “learnability” of a class and ease-of-use of the library. Indeed, if options are arguments of a routine, one needs to learn all arguments (understand what they are used for) before being able to use the routine, whereas it would be much easier to learn only the operands and then understand options one at a time, when one actually needs it.

11. Design patterns (5 points)

Question 19: Prototype (5 points)

19.1 What is the goal of the Prototype pattern? (2.5 points)
19.2 How would you write it in Eiffel? (2.5 points)

19.1 The goal of the Prototype pattern is to specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

19.2 In Eiffel, there is no need for a special design to satisfy the intent of the Prototype pattern, because any class inherits from the universal class `ANY` which has cloning facilities (features `clone` and `deep_clone`).

12. Concurrency (10 points)

Question 20: Concurrency (10 points)

20.1 Write a generic, concurrent class `CONCURRENT_STACK [G]` for stacks. Reuse the already existing sequential class `STACK [G]` (see below).

Hint: `CONCURRENT_STACK [G]` should inherit from `STACK [G]`. (3 points)
class interface STACK[G]

feature -- Element change

    put (x: G)
    -- Add x to beginning.
    require
        not_full: not is_full
    ensure
        not_empty: not is_empty
        one_more: count = old count + 1

    remove
    -- Remove item on top.
    require
        not_empty: not is_empty
    ensure
        not_full: not is_full
        one_less: count = old count - 1

feature -- Access

    item: G
    -- Item at top
    require
        not_empty: not is_empty
    ensure
        not_changed: count = old count

feature -- Status report

    is_empty: BOOLEAN
    -- Is there no element?
    ensure
        definition: Result = (count = 0)

    is_full: BOOLEAN
    -- Is the data structure filled to capacity?
    ensure
        definition: Result = (count = capacity)

    count: INTEGER
    -- Number of items

    capacity: INTEGER
    -- Maximum number of elements the array can contain

invariant

    count_positive: count >= 0
    consistent: count <= capacity

end

20.2 Encapsulate the access to concurrent stacks by creating another class
CONCURRENT_STACK_ACCESS. (4 points)
20.3 Give an example how to use the both classes. (3 points)

20.1

**separate class** CONCURRENT_STACK [G]

*inherit*

STACK [G]

*end*

20.2

**class** CONCURRENT_STACK_ACCESS [G]

*feature* -- Element change

  *put* (cs: CONCURRENT_STACK [G]; x: G) is
  -- Put x in in cs,
  -- waiting if necessary until there is room.
  *require*
  cs_not_void: cs /= Void
  not_full: not cs.is_full
  *do*
  cs.put (x)
  *ensure*
  not_empty: not cs.is_empty
  *end*

  *remove* (cs: CONCURRENT_STACK [G]) is
  -- Remove the last element from cs, waiting if
  -- necessary until there is such an element.
  *require*
  cs_not_void: cs /= Void
  not_empty: not cs.is_empty
  *do*
  cs.remove
  *ensure*
  not_full: not cs.is_full
  *end*

  *item* (cs: CONCURRENT_STACK [G]): G is
  -- Last item in cs (LIFO)
  *require*
  cs_not_void: cs /= Void
  not_empty: not cs.is_empty
  *do*
  Result := cs.item
  *end*

*end*
20.3

test is
   -- Use classes CONCURRENT_STACK and
   -- CONCURRENT_STACK_ACCESS.

local
    cs: CONCURRENT_STACK [INTEGER]
    csa: CONCURRENT_STACK_ACCESS [INTEGER]
    res: INTEGER

do
   create cs
   create csa
   csa.put (cs, 15)
   csa.put (cs, 25)
   res := csa.item (cs)
   io.put_integer (res)
   io.new_line
   csa.remove (cs)
   res := csa.item (cs)
   io.put_integer (res)
end