Exercise 4

Hand-out: 26 April 2005
Due: 11 May 2005

Please solve this exercise within your project team.

1. About your project

1) When working on your project, please note that it is not accepted to use copyrighted materials in your game. This means that you will have to use game materials, for example, images that are not copyrighted. Or you can draw them yourselves.

2) Make sure with your CVS account you are working for your team project on the required folder structure according to the previous exercise notes, and
   - No offensive content!
   - Use all lowercase letters for your filenames including the suffix part, e.g. *.JPG should be *.jpg. This will allow you to compile your project on other platforms too.
   - Use only slashes "/" in path names, which is also for platform independence.

2. 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).

Below is extracted from the textbook (page 167), which is an example 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.

For each use of the word “object”, “thing” or “user” in the above text, underline the word if you think that the authors really meant “object”; double-underline the word if you think that they really meant “class”.

1
3. Design by Contract

Below is the ADT specification of an unbounded queue (FIFO, First-In First-Out):

<table>
<thead>
<tr>
<th>TYPES</th>
</tr>
</thead>
<tbody>
<tr>
<td>- ( \text{QUEUE} [G] )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FUNCTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>- ( \text{put} ): ( \text{QUEUE} [G] \times G \rightarrow \text{QUEUE} [G] )</td>
</tr>
<tr>
<td>- ( \text{remove} ): ( \text{QUEUE} [G] \rightarrow \text{QUEUE} [G] )</td>
</tr>
<tr>
<td>- ( \text{item} ): ( \text{QUEUE} [G] \rightarrow G )</td>
</tr>
<tr>
<td>- ( \text{empty} ): ( \text{QUEUE} [G] \rightarrow \text{BOOLEAN} )</td>
</tr>
<tr>
<td>- ( \text{new} ): ( \text{QUEUE} [G] )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AXIOMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>For any ( x: G, q: \text{QUEUE} [G] )</td>
</tr>
<tr>
<td>- ( \text{item} (\text{put} (q, x)) = \begin{cases} \text{item} (q) \text{ if not empty} (q) \ x \text{ if empty} (q) \end{cases} )</td>
</tr>
<tr>
<td>- ( \text{remove} (\text{put} (q, x)) = \begin{cases} \text{put} (\text{remove} (q, x)) \text{ if not empty} (q) \ q \text{ if empty} (q) \end{cases} )</td>
</tr>
<tr>
<td>- ( \text{empty} (\text{new}) )</td>
</tr>
<tr>
<td>- ( \text{not empty} (\text{put} (q, x)) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PRECONDITIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>- ( \text{remove} (q: \text{QUEUE} [G]) ) \text{ require not empty} (q)</td>
</tr>
<tr>
<td>- ( \text{item} (q: \text{QUEUE} [G]) ) \text{ require not empty} (q)</td>
</tr>
</tbody>
</table>

\( \text{QUEUE} [G] \) denotes a generic class (see page 320-325 in the textbook), and it represents a set of possible types (for example \( \text{QUEUE} [\text{INTEGER}] \), \( \text{QUEUE} [\text{PROCESS}] \), etc.). The text of class \( \text{QUEUE} [G] \) can use \( G \) as a common representation for concrete generic parameters such as \( \text{INTEGER} \) or \( \text{PROCESS} \).

Write the interface of the Eiffel class \( \text{QUEUE} [G] \) corresponding to the above ADT.

4. Project Design

The project design is the last phase before you can start to implement. It is a crucial phase of the project planning, since a lot of decisions have to be made.
4.1 The Classes

Your project, like every other object oriented program, will consist of many different classes. The way you break down your project into these classes is going to have a major influence on the rest of the development and implementation.

Break down your project into classes:

Some suggestions:
1. A class represents a set of objects
2. Class-names are almost always nouns (exception: behavioral classes)
3. In some cases classes model actions, e.g. command classes whose main purpose is to execute something. Such classes should still have nouns as names (e.g. OPEN_FILE_ACTION)
4. Introduce deferred classes
5. Avoid “taxomania“ (over classification see: OOSC 2: 24.4)

4.2 The BON-Diagram

Class diagrams are used in nearly all Object-Oriented software designs. They are used to describe the classes of the system and their relationships to each other. Diagrams make use of the fact, that the humans do elaborate pattern matching using their eyes. It is therefore easier to get an understanding of the whole project design by looking at a diagram then by reading thousands of lines of code.

According to your class breakdown, provide a BON-diagram design of the project.

Please stick to the standard!

Some suggestions:
1. Use a tool to draw the BON-diagram
   - BON-diagrams can be drawn using Eiffel Studio University Edition which is installed on the student's computer in the IFW c31 and d31.
2. Omit uninteresting facts (e.g. Every class inherits form ANY)

The BON Standard

- Classes are symbolized by a yellow ellipsis with a black border. The name of the class is written inside the ellipsis in a blue font. Generic parameters belong to the class name.
- In order to symbolize deferred classes an asterisk (*) is put into the top of the ellipsis, a plus (+) denotes an effective class. The root class has a double outline.
- In larger projects, classes are grouped into clusters. In the BON-diagram a dashed line is used to surround all classes of a cluster. And the name of the cluster is written right next to it.
- One of the main aspects of object oriented programming is the concept of inheritance.
- A red arrow is used to show inheritance. The head of the arrow points at the parent class.
- Sometimes a client–supplier relation may also be important to understand the design of the software. To show such a relation a blue double lined arrow, which points at the supplier class, is used. Usually the name of the feature is written beside the arrow.

![Figure 4.2  illustration of the BON-diagram](image)

For more details see OOSC2: 27.7

### 4.3 The Interface

In order to get a complete picture of the design and to understand how the classes work together, it is necessary to specify the interfaces. This is also useful if different developer work on different classes which depend on each other.

- For each of your classes write down the public interface.
- Complete your BON-diagram with the most important client–supplier relations. But don't overcrowd the diagram.

Some suggestions:
1. A Interface consist of:
   - The class name
   - All inherited classes
   - All visible features including: header comment, preconditions and postconditions
   - All invariants
2. Be careful when you design an interface, since it can be very time consuming to commit changes to an interface in a late phase of the implementation.