2024/2025 EXAM GUIDE

CONTENTS

ENTERING THE EXAMINATION

- Maximum number of seminar absences allowed: **4.**
- Maximum number of laboratory absences allowed: **2.**
- If you do not have the minimum number of attendances, you cannot enter examination during the **regular, or retake** session during 2024.
- Minimum laboratory grade to enter examination during regular session: **5.00** (no rounding).

STUDENTS WHO CANNOT TAKE PART IN EXAMINATION

- In case your lab grade is < **5.00** you may only attend the retake examination.
- In case you have not paid your school taxes, you are not allowed to take the exam (check the latest announcements on the faculty website).

- **Make sure to verify your situation and notify us of any issues as soon as possible!**

OBJECTIVE AND COURSE CONTENTS

OBJECTIVES

- Being familiar with some fundamental concepts in computer programming
- Introduction to basic concepts in software engineering
- Learn elements of software design, architecture, implementation and maintenance
- Familiarization with some of the software tools used in large-scale application development (unit testing, coverage)
- Basic knowledge of the Python 3.x programming language
- Using the Python language in software development, testing, running and debugging applications
- Learning/honing your own programming style ☺

COURSE CONTENT

PROGRAMMING IN THE SMALL

1. Introduction to the Python 3 Language
2. Recursion. Computational Complexity
3. Searching. Sorting
4. Problem Solving Methods

PROGRAMMING IN THE LARGE

5. Procedural programming
6. Modular Programming
7. Unit Tests. Exceptions
8. Classes and Objects
9. Design Principles for Modular Programs
10. UML Class Diagrams
11. Using Text and Binary Files
12. Layered architecture. Inheritance.
13. Testing.

EVALUATION

DURING THE SEMESTER

[L] 40% Laboratory – grade based on your activity during the semester

- 50% - Weighted average of 10 lab assignments, with 0 for the labs you were not graded for.
- 50% - Weighted average of grades obtained in two tests taken during the laboratory (20% first one, 30% second one).
- **NB!** The lab grade must be **>=5.00** (***2 decimals, no rounding***) to enter the written/practical examination during the regular exam session

[L_B] 0 - ~1p Laboratory assignment bonus – optional bonus for extra laboratory work. This is added to the laboratory grade.

[S_B] 0 – ~0.5p Seminar bonus – optional bonus for your activity during the seminar. This is added to the laboratory grade.

DURING THE EXAM SESSION

[W] 30% Written exam – on the examination date.

- The written exam grade must be **>=5.00** (***2 decimals, no rounding***) to pass the course.

[P] 30% Practical exam – on the examination date, after the written exam

- The practical exam grade must be **>=5.00** (***2 decimals, no rounding***) to pass the course.

Final grade: $0.4 * \min(10, L + S_B + L_B) + 0.3 * W + 0.3 * P$

THE RETAKE SESSION

- During the retake session you may hand in laboratory work but are limited to a maximum laboratory grade [**L**] of **5.00**.
- You may choose to retake the written, the practical, or both examinations in case you have failed/not attended during the regular examination session.
- If you want to increase the grade obtained during the regular session, you may partake during the retake session. Your final grade will be the largest one between those obtained at each part of the test.

EXAMINATION DATES

- Examination dates are set in the ***Academic Info*** application and published on the ***General*** channel of the course's Team.
- Attending on the backup date is possible only with the agreement of the course coordinator and for reasons proven with documentation.

Important

- Make sure you've fulfilled your financial obligations towards the University, otherwise we are not allowed to grade you.
- Re-check the date/time of the exam beforehand (MS Teams, @General channel announcements as well as any messages on your own group's subchannel)
- Be present on time and **have a photo ID ready**
- Have your computer ready with Python 3.x and your preferred IDE installed.
- It is possible that the problem statement during the practical exam will require drawing a board ☺. You are allowed to install the texttable Python library that provides this functionality. You can find the component here:
    - https://github.com/foutaise/texttable
    - **NB**! Read how to install it and make sure you know how to use it.
- During the practical exam, you are allowed use of the following libraries:
    - Those included in the default Python installation.
    - texttable or an equivalent library, used only to draw tables/boards.

After the exam

- We will grade your work as soon as possible.
- You must check your grades in the AcademicInfo application and report any errors ASAP!
- **Errors cannot be corrected after the examination session is complete.**

WHAT THE EXAMINATION COVERS

BASIC ELEMENTS OF THE PYTHON 3 LANGUAGE

- Instructions: ***=, ==, if, while, for.***
- Predefined data types: ***integer, real, string, list, dictionary, tuple.***
- Functions: ***defining, parameter transmission, specification.***
- User defined types – ***classes, objects, (static) methods, attributes, inheritance.***
- Exceptions – defining ***exception types, raising, catching***.
- Lambda expressions.

ALGORITHMS – SPECIFICATION/TESTS/IMPLEMENTATION

Possibilities:

- You are given the specification – implement and test
- You are given the implementation – specify and test it
- You are given a test function – implement and specify the function it tests

1. Implement and test the function having the following specification

```
"""
    Compute the sum of even elements in the given list
    input:
        l - the list of numbers

    output:
        The sum of the even elements in the list

    Raises TypeError if parameter l is not a Python list
    Raises ValueError if the list does not contain even numbers
"""
```

2. Specify and test the following function

```
def function(n):
    d = 2
    while (d < n - 1) and n % d > 0:
        d += 1
    return d >= n - 1
```

ALGORITHM COMPLEXITY

You are given a function – analyze its complexity (best case, average case, worst case) as well as the extra-space complexity

3. Analyze the runtime complexity for the following function

```
def complexity_1(x : list):
    m = len(x)
    found = False
    while m >= 1:
        c = m - m / 3 * 3
        if c == 1:
            found = True
        m = m / 3
```

4. Analyze the runtime complexity for the following function:

```
def complexity_2(x : list):
    found = False
    n = len(x) - 1
    while n != 0 and not found:
        if x[n] == 7:
            found = True
        else:
            n = n - 1
    return found
```

5. Analyze the runtime complexity for the following function:

```
def complexity_3(n : int, i : int):
    if n > 1:
        i *= 2
        m = n // 2
        complexity_3(m, i - 2)
        complexity_3(m, i - 1)
        complexity_3(m, i + 2)
        complexity_3(m, i + 1)
    else:
        print(i)
```

6. Create an iterable data structure and a ***Product*** class with attributes ***name***, ***type*** and ***price.*** Write a generic sort function having `n*log(n)` time complexity. Create an instance of the iterable data structure and add 10 products to it. Use your sort function implementation to sort the list:

- Alphabetically by product name
- Decreasing by price

PROGRAMMING TECHNIQUES

Studied during this course:

- Divide and conquer
- Backtracking
- Greedy
- Dynamic programming

Possibilities:

- You are given a problem statement with a solution within one of the given techniques.
- Select the adequate technique for solving a given problem statement
- What we will ask:
    - Indicate the solution in detail (with or without implementation).
    - ***Backtracking:***
        - ***Search space*** representation***, consistent***() and ***solution***() functions
    - ***Divide and conquer:***
        - ***Divide*** – description, explain why you choose to do it that way
        - ***Conquer –*** describe how it works
        - ***Combine –*** describe how it works
    - ***Greedy method:***
        - Describe the ***set of candidates***
        - Describe how you make each ***selection***
        - Describe how you ***update*** the set of candidates
    - ***Dynamic programming:***
        - How was the ***principle of optimality*** observed.
        - The recurrence describing the algorithm.
        - Overlapping ***sub-problems***.
        - How you used ***memoization***.

10. Determine the longest subsequence of decreasing numbers in a list using dynamic programming.

11. Select the most appropriate technique and describe the solution for calculating the sum of the even numbers in a list.

12. Use the divide and conquer technique to calculate the product of the elements that are found on index positions multiple of 3 in a list (e.g., these are positions 0, 3, 6 and so on).

13. You are given an ***n*** × ***n*** checkerboard, where each cell contains a non-negative integer value representing a reward. The goal is to start from the top-left corner of the checkerboard and traverse to the bottom-right corner, collecting the maximum reward possible along the way. However, there are constraints on the allowed moves: from any cell, you can only move **right** or **down**.

More learning resources and examples for problem solving methods:
https://www.cs.princeton.edu/~wayne/kleinberg-tardos/ (relevant sections on left-side menu)

PRACTICAL EXAMINATION

Below you will find problem statements similar to what you can expect as part of the practical exam. The problem statement will in general follow the requirements set out between ***Assignment 7*** and ***Assignment 10***, will require writing specifications, tests and an implementation using layered architecture, classes and objects.

Observations:

1. Solving the following problem statement completely should be possible for you in a timespan between 3 and 4 hours, as it has a longer list of requirements.
2. In order to pass the practical exam, you must ***implement a working program!***
3. Aspects that were part of bonus points (e.g. GUI, SQL-backend, minimax) are not required.

PROBLEM STATEMENT – STUDENT ASSIGNMENTS

Write an application to help with the management of laboratory activities for a faculty course such as FP. Students enrolled in the class can be assigned one of the **20 problem statements** (numbered from **1 to 20**) from each **laboratory**, and when they turn it in they are graded. The application will be used by the teacher and will provide the following functionalities:

- Add a student to the course. Each student has an ***id***, a ***name*** and a ***group***. You cannot have more than one student having the same ***id***, as well as students without a name or group.
- Remove a student from the course. A student can only be removed while they have not received any grades.
- Assign a laboratory problem statement to a student. You cannot assign more than one problem statements from the same laboratory to a student. If the student was already assigned a problem, the program must report the error.
- Assign a laboratory to a group. Each student in the group will be assigned a problem statement. Implement an algorithm to assign students with problem statements (e.g. each subsequent student is assigned the next problem in the list of problem statements). In case a student was already assigned a problem statement, this must not be changed!
- Grade a student for a laboratory, with an integer grade 1 to 10. Validate that the grade is valid. Grades cannot be changed!
- Best/worst students in a group. Given a group, list its students in decreasing order of their average grade.
- Students failing the class. Provide the list of all the students (regardless of group), for whom the average grade is less than 5.
- Undo/redo the last performed operation that changes the list of students or grade assignments.

NB! Data is **loaded** and **saved** to two text files: “***students.txt***”, “***grades.txt***”. When starting the program, make sure to have at least 10 students in the students file.

NB! Solutions without file-based repositories are acceptable, but the maximum grade for such solutions is ***7***.

NB! Your solution must adhere to the principles of layered architecture studied during the course. You will have domain, repository, controller, and UI packages/modules. The diagram below provides a guideline regarding the implementation, but it is not meant to be exhaustive.

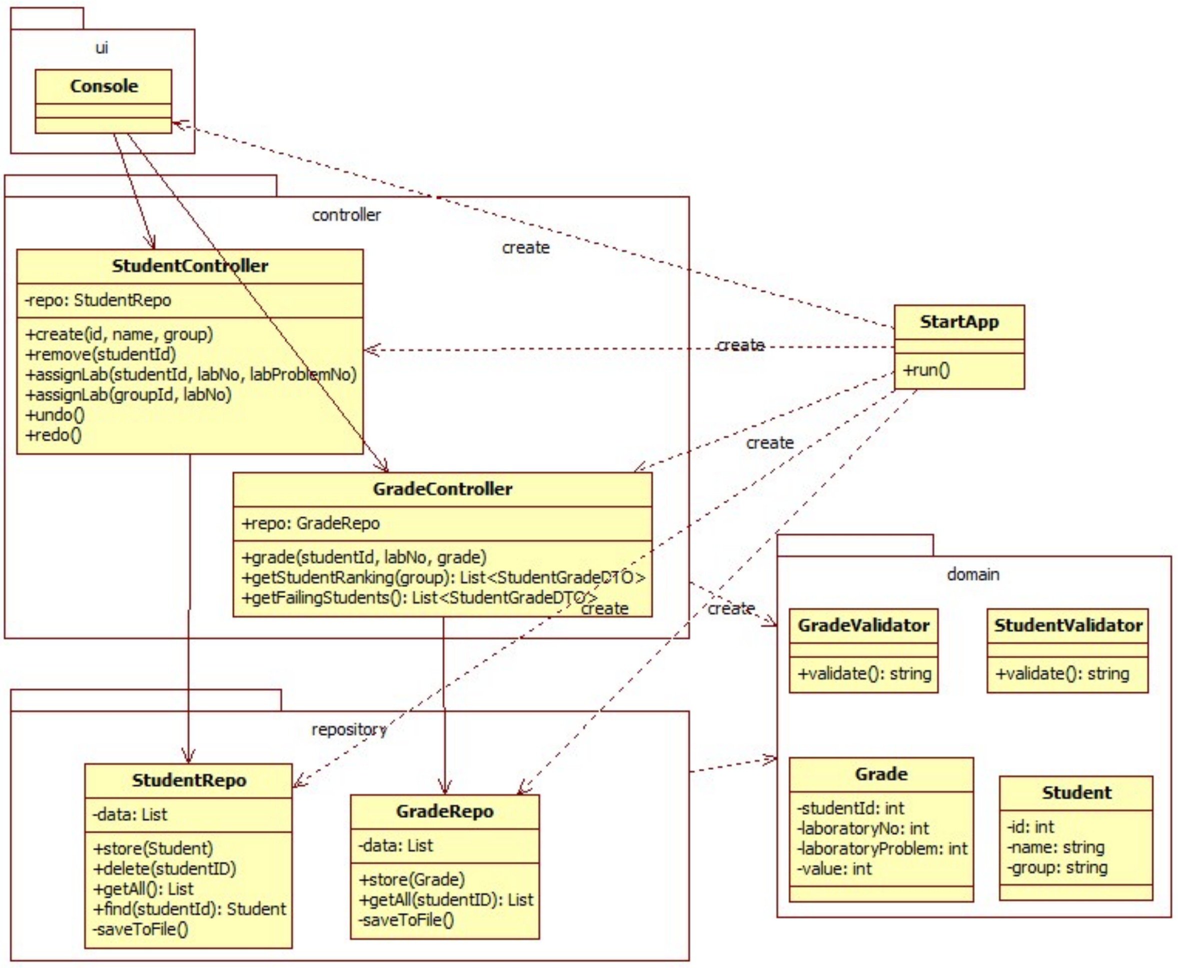

FIGURE 1 - CLASS DIAGRAM

Babeș-Bolyai University
Cluj Napoca
Faculty of Mathematics and Computer Science
Fundamentals of Programming

PROBLEM STATEMENT - BATTLESHIP[1]

Implement a console-based variation of the classical board game that you can play against the computer. To keep things simple, the game is played on a 6x6 grid as shown in the figure below. Before the game starts, both players place two battleships on the board, so that no part of the ships is outside the board and they do not overlap. Once the game starts, players take turns attacking squares, with hits tracked on the player and targeting boards. The game ends when one player hits all the squares occupied by the enemy ships. Program functionality is broken down as follows:

1. Place your battleships on the board using the following command: **`ship`** **<$C_1L_1C_2L_2C_3L_3$>**
   [E.g. commands **`ship C3D3E3`** and **`ship A0A1A2`** gives the ship position in the figure below]
   In case an invalid square is provided, a part of the ship falls outside the grid, or ships overlap (have at least one common square) the program will provide an error message and will not place the ship **[2p]**.

2. You can repeat the **`ship`** command as many times as you wish, until you are pleased with your ships' position on the grid. If you already placed two ships on the board, entering a valid command will replace the ship that was added first with the current one. **[1p]**

3. Each time the command results in valid placement of a ship on the player's board, the player board will be displayed as illustrated on the left hand side of the figure below. **[1p]**.

4. Start the game using the following command: **`start`**
   The start command can only be provided once two battleships have been placed on the player board. Once the game starts, the program will randomly place two battleships on the computer's board, using the same rules. **[2p]**

```
  A B C D E F       A B C D E F
0 + . . . . .     0 . . . . . .
1 + . . . . .     1 . . . . . .
2 + . . . . .     2 . . . . . .
3 . . + + + .     3 . . . . . .
4 . . . . . .     4 . . . . . .
5 . . . . . .     5 . . . . . .
  Player board    Targeting board
```

5. Play the game **[2p]**. The player and computer will attack squares in turn, with the player having the first attack. Attacks are made using the command: **`attack`** **<*square*>**. [E.g. **`attack E4`**]. When all the squares containing a player's ships are hit, the player loses the game. The program will provide a message in this regard.

   After each attack, the player and targeting boards are updated. Given the player and targeting boards above, the following series of attacks will result in the following possible board configuration:

```
attack E4                    A B C D E F       A B C D E F
Player misses!             0 + . . . . .     0 . . . . . .
computer attack C2         1 + . . . . .     1 . . . . . .
Computer misses!           2 + . o . . .     2 . . X . . .
attack C3                  3 . . + X + .     3 . . X . . .
Player hits!               4 . . . . . .     4 . . . . o .
computer attack D3         5 . . . . . .     5 . . . . . .
Computer hits!               Player board     Targeting board
attack C2
Player hits!
```

6. Cheats. We want to know where the computer's battleships are. Using the **`cheat`** command, the program will reveal the placement of the computer's ships (it is up to you to decide how) **[1p]**.

Non-functional requirements:

- Implement a layered architecture solution.
- Provide specification and tests for the methods involved in functionalities 1 and 2. In case no specifications or tests are provided, these functionalities are graded at 50% value.

[1] Actual subject during the 2016/2017 retake exam ☺

Babeș-Bolyai University
Cluj Napoca
Faculty of Mathematics and Computer Science
Fundamentals of Programming

PROBLEM STATEMENT - AIR TRAFFIC CONTROL[2]

Write a console-based application to help air traffic control (ATC) monitor all domestic flights taking place during one day (00:00 – 23:59). The application must include the following features:

1. Flight information is kept in a text file, using the format in the example below. When the program starts, flight information is read from the file **[1p]**. Each modification is persisted to the text file **[1p]**.
2. Add a new flight. Each flight has an ***identifier***, a ***departure city*** and ***time***, and an ***arrival city*** and ***time*** **[1p]**. Flight identifiers are unique; flight times are between 15 and 90 minutes; an airport can handle a single operation (departure or arrival) during each minute **[1p]**.
3. Delete a flight. The user provides the flight identifier. If it does not exist, an error message is displayed **[1p]**.
4. List the airports, in decreasing order of activity (number of departures and arrivals during the day) **[1p]**.
5. List the time intervals during which no flights are taking place, in decreasing order of length. **[1.5p]**.
6. The tracking radar suffers a failure. The backup radar can be used, but it can only track a single flight at a time. Determine the maximum number of flights that can proceed as planned. List them using the format below **[1.5p]**:

```
05:45 | 06:40 | RO650 | Cluj - Bucuresti
```

Non-functional requirements:

- Implement an object-oriented, layered architecture solution using the Python language.
- Provide specification and unit tests for Repository/Controller functions related with the ***second functionality***. In case specification or tests are missing, the functionality is graded at 50%.

Observations!

- The day starts at 00:00 and ends at 23:59.
- ***Default 1p***.

Example input file.

```
RO650,Cluj,05:45,Bucuresti,06:40
0B3302,Cluj,07:15,Bucuresti,08:15
SLD322,Timisoara,09:05,Cluj,10:00
RO643,Bucuresti,10:15,Cluj,11:10
RO734,Timisoara,10:45,Iasi,12:25
KL2710,Timisoara,14:25,Bucuresti,15:40
RO745,Cluj,12:50,Iasi,14:05
RO746,Iasi,14:30,Cluj,15:50
RO647,Bucuresti,18:05,Cluj,19:00
KL2706,Bucuresti,18:10,Timisoara,19:05
RO733,Iasi,08:30,Timisoara,10:20
SLD363,Cluj,19:35,Timisoara,20:30
RO649,Bucuresti,21:55,Cluj,22:50
0B3101,Bucuresti,22:55,Cluj,23:55
RP621,Bucuresti,07:30,Oradea,08:55
RO622,Oradea,09:20,Bucuresti,10:40
RO627,Bucuresti,17:55,Oradea,19:20
RO628,Oradea,19:45,Bucuresti,21:05
XL897,TgMures,08:55,Oradea,09:30
XL898,Oradea,20:45,TgMures,21:25
LH012,Iasi,14:35,Oradea,15:35
LH013,Oradea,17:00,Iasi,18:10
```

[2] Actual subject during the past few years' exam sessions ☺

PROBLEM STATEMENT - HANGMAN[3]

You have to implement a console-based variation of the classical Hangman game. The computer will select a **Sentence** that the user can attempt to guess, letter by letter. Each time the user guesses a correct letter, the computer will fill it in the sentence at the correct positions. In case the letter does not appear, the computer will fill in a new letter in the word "hangman", starting from the empty string. The game ends when the user has guessed the sentence (user wins) or when the computer fills in the "hangman" word (user loses). Program functionality is broken down as follows:

1. Sentences are stored in a text file that is read when the program is started **[1p]**.
2. Add a sentence. While not in a game, the user can add a sentence that is stored in the text file. **[1p]**. Each sentence must consist of at least 1 word. Every word in the sentence must have at least 3 letters. There can be no duplicate sentences. **[1p].**
3. Start the game. When the user starts a game, the computer selects one of the available sentences and displays it on screen, hangman-style. This means that the computer reveals the first and last letter of every word, as well as all the apparitions of these letters within the words **[2p]**. The sentence selection is random **[1p]**.
   e.g. for the sentence "**anna has apples**", the computer reveals "**a _ _ a has a _ _ _ _ s**".
4. Play the game. The game consists of several rounds. In each round, the user proposes a letter. If the sentence contains the letter, the computer reveals where these letters appear within the sentence. If the sentence does not contain the letter, or the user previously proposed the letter the computer will add a new letter to the word "hangman", which is displayed to the user **[2p].**
5. Game over. The game is over when the sentence is correctly filled in (user wins), or when the computer fills in the word "hangman" (user loses). **[1p]**

Non-functional requirements:

- Implement an object-oriented, layered architecture solution using the Python language.
- Write tests and specifications for the second functionality. **Functionality 2 is not graded otherwise**.

Observations!

- Sentences are loaded from/saved to a text-file that must initially hold at least 5 entries.
- Default **1p**.

sentences.txt example:

```
anna has apples
patricia has pears
cars are fast
planes are quick
the quick brown fox jumps over the lazy dog
```

Gameplay example:

```
Output: "a _ _ a has a _ _ _ _ s" - ""
User guess: "m", output changes to: "a _ _ a has a _ _ _ _ s" - "h"
User guess: "n", output changes to: "anna has a _ _ _ _ s" - "h"
User guess: "m", output changes to: "anna has a _ _ _ _ s" - "ha"
User guess: "e", output changes to: "anna has a _ _ _ es" - "ha"
User guess: "x", output changes to: "anna has a _ _ _ es" - "han"
User guess: "t", output changes to: "anna has a _ _ _ es" - "hang"
User guess: "p", output changes to: "anna has app _ es" - "hang"
User guess: "l", output changes to: "anna has apples" - "YOU WON!"
```

[3] Actual subject during the past few years' exam sessions ☺

PROBLEM STATEMENT - TRAIN TRAVEL

You must create a console-based application that helps a travel agency sell train tickets. The agency has several **Routes** it has tickets for and wants to keep track of the number of available seats for each route as well as their income from selling tickets. The income is obtained by adding the prices for all tickets sold. The price of train tickets is based on a flat hourly rate and is calculated as the product between the hourly rate and the travel time of the route, with minute precision. The functionalities that you must implement are:

1. Add a new train route. Each train route has a unique ***number***, a ***departure city*** and ***time***, an ***arrival city*** and ***time***, and the ***number of available tickets***. The departure and arrival cities must be distinct, the departure time must precede the arrival time (assume the same day), and the number of available tickets must be a positive integer (***1.5p***).
2. Sell a ticket. The user enters the train number, and the program calculates and displays the price of the ticket. If the user accepts, the price is added to the company's income and the number of available seats on the route is decreased by 1. If the user declines, the number of seats available is not changed (***1.5p***).
3. Show the total income. This will show the total income on-screen (***1p***).
4. Show report. This will show an ordered list of train routes, sorted in descending order by the number of tickets sold for each route. The number of tickets sold must also be displayed (**2*p***).

Non-functional requirements:

- Implement a layered architecture solution with Repository, Controller and UI (***1.5p***).
- Provide specification and tests for all non-UI methods (***1.5p***).

Observations!

- The list of train routes is loaded from/saved to a text-file that must initially hold at least 5 entries.
- Solutions that store train routes in memory are graded at 50% for all functional requirement (maximum is 3p).
- The exam cannot be passed without working functionalities!
- The first number in the example below is the value of the flat hourly rate

routes.txt example:

```
15
1742,Satu Mare,10:20,Bucuresti,20:30,100
1743,Bistrita,12:00,Cluj,14:15,50
1856,Constanta,00:00,Bucuresti,03:30,220
1900,Tulcea,05:30,Galati,06:45,90
1901,Galati,09:00,Tulcea,10:15,100
```