Examination Guide

Contents

1. Objectives

- Analyse, design and implement solutions for problems in defined professional contexts.
- Create projects with clear separations on architectural layers, based on different architectural patterns.
- Apply design patterns in different contexts.
- Write medium/small scale Java programs (including graphical user interfaces in Java).

2. Course Contents

1. Software analysis and design.
2. Introduction in Java.
3. Classes, inheritance, interfaces, packages, exceptions.
4. Generic types, collections.
5. Test Driven Development.
6. I/O in Java.
7. Java Database Connectivity API.
8. Java 8 features.
9. Graphical User Interfaces: JavaFX.
10. Java Reflection API.
11. Concurrency.
12. Design Patterns.
13. The .NET platform. The C# programming language.
14. Revision.

3. Evaluation

3.1. During the semester

Lab grade – **L (40% of the final grade)**:

Mathematics Computer Science

- 70%: Lab assignments: weighted arithmetic mean of 5 lab and home assignments, with 0 for the labs you did not present.
- 30%: the practical test.

Artificial Intelligence

- 50%: Lab assignments: weighted arithmetic mean of 9 lab and home assignments, with 0 for the labs you did not present.
- 50%: the practical tests (25% for each test).

Seminar bonus – **SB**: 0 – 0.5 – optional bonus for your activity during the seminars. **This is added to the final grade.**

Practical/laboratory bonus – **B**: optional bonus for outstanding solution to the laboratory problem or for an additional requirement. **This is added to the final grade.**

3.2. During the examination session

Written examination – **W (30% of the final grade)**: on your examination date.

Practical examination – **P (30% of the final grade)**: on your examination date.

Final Grade: G = 0.4 x L + 0.3 x P + 0.3 x W + SB + B

To pass the examination grades W, P and G must be ≥ 5 (no rounding).

3.3. During the retake session

- The final grade in the retake session is computed by the same algorithm presented above.
- You can choose to retake the written, practical, or both examinations in case you have failed/not attended during the regular examination session.
- If you want to increase the grade you obtained during the regular session, you may participate to the examinations during the retake session. Your final grade will be the largest one between those obtained.

4. Examination dates

Mathematics Computer Science

16.01.2025 – groups 821, 822 – written examination (during the lecture)

10.01.2025 – group 821/2, 822/2 – practical examination (during the laboratory)

16.01.2025 – group 821/1 – practical examination (during the laboratory)

17.01.2025 – group 822/1 – practical examination (during the laboratory)

18.02.2025 – groups 821, 822 – written and practical examinations (retake session)

Artificial Intelligence

	Main date	**Backup date**	**Retake date**
	07.02.2025	**06.02.2025**	**18.02.2025**
Time for written exam	09:00	09:00	09:00
Room(s) for written exam	L001, L002	L338	L320, L321
Time for practical exam	11:30	11:30	11:30
Rooms for practical exam	L336, L338, L339	L306	L320, L321

Important observations!

- 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.
- Arrive on time.
- Bring your laptop for the practical examination.

5. Written Examination

Allotted time: ~ 1 hour.

Determine the result of the execution of the following Java programs. Justify your answers. If there are any errors, signal them, correct them and explain the result after the error has been corrected. We assume all static functions are defined in a **Main** class and are called in a public static function *main*(). **(1p each)**

```
interface Shape {
    public double getArea();
}

interface Polygon extends Shape {
    public double getArea();
}

class Rectangle implements Polygon {
    private int length, width;

    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }

    public double getArea() {
        return this.length * this.width;
    }

    @Override
    public String toString() {
        return "Rectangle: " + getArea();
    }
}

class Square extends Rectangle {
    public Square(int length) {
        super(length, length);
    }

    @Override
    public String toString() {
        return "Square: " + getArea();
    }
}

class Circle implements Shape
{
    private int radius;

    public Circle(int radius) {
        this.radius = radius;
    }

    public double getArea() { return Math.PI *
Math.pow(radius, 2); }

    @Override
    public String toString() {
        return "Circle: " + getArea();
    }
}
```

```
class AreaCalculator<T extends Polygon> {
    private T[] elements;

    public AreaCalculator(T[] elements) {
        this.elements = elements;
    }

    double[] computeAreas() {
        double[] res = new double[elements.length];
        int i = 0;
        for (T e: elements) {
            res[i++] = e.getArea();
        }
        return res;
    }
}
```

Consider PI to be 3.14.

A

```
public static void function1() {
    Polygon r1 = new Rectangle(2, 3);
    Rectangle r2 = new Square(3);

    System.out.println(r1 instanceof
Rectangle);
    System.out.println(r2 instanceof
Rectangle);
    System.out.println(r1 instanceof
Square);
    System.out.println(r2 instanceof
Polygon);
    System.out.println(r2.getArea());
}
```

B

```
public static void
function2_aux1(List<Polygon> l){
    for (Polygon p: l)

System.out.println(p.getArea());
}

public static void
function2_aux2(Polygon ... l) {
    for (Polygon p: l)

System.out.println(p.getArea());
}


public static void function2() {
        List<Square> l = new
ArrayList<>();
        Square s1 = new Square(2);
        Square s2 = new Square(3);
        l.add(s1);
        l.add(s2);
        function2_aux1(l);
        function2_aux2(s1, s2);
    }
```

C

```
public static void function3(){
    Square[] squares = {new Square(2), new Square(1), new Square(5)};
    AreaCalculator<Square> calculator1 = new AreaCalculator<>(squares);
    double[] areasSquares = calculator1.computeAreas();
    for (double a: areasSquares)
        System.out.println(a);

    Circle[] circles = {new Circle(1), new Circle(2)};
    AreaCalculator<Circle> calculator2 = new AreaCalculator<>(circles);
    double[] areasCircles = calculator2.computeAreas();
    for (double a: areasCircles)
        System.out.println(a);
}
```

D

```
public static void function4() {
    ArrayList<Shape> shapes = new ArrayList<>(Arrays.asList(new Rectangle(1, 2), new
Square(1), new Square(2), new Circle(1)));
    shapes.stream().filter(s -> s instanceof Polygon).sorted(new Comparator<Shape>() {
        @Override
        public int compare(Shape o1, Shape o2) {
            if (o1.getArea() > o2.getArea())
                return -1;
            else if (o1.getArea() < o2.getArea())
                return 1;
            else
                return 0;
        }
    }).forEach(System.out::println);
}
```

E

```
@FunctionalInterface
interface StringProcess {
    String process(String s);
}

public static void function5() {
    StringProcess process1 = s -> s.toUpperCase();
    StringProcess process2 = s -> {
        String[] parts = s.split(" ");
        String newString = "";
        for (String part : parts){
            newString = newString + part.substring(0, 1).toUpperCase() +
part.substring(1).toLowerCase();
        }
        return newString;
    };

    System.out.println(process1.process("welcome to the examination"));
    System.out.println(process2.process("welcome to the examination"));
}
```

Design and implement in Java an object-oriented system that allows automatic checking of book sections. A section has a title, a content and aggregates a checker. Furthermore, a section has a function *generate()* which prints the title and content only if the section is correct from the point of view of the checker. Checking will be done by an abstract *Checker*, which contains a function *check(s: Section)* that returns a boolean value. Some concrete checkers will have to be implemented (see below).

1. Implement the class *Section*. **(1p)**
2. Implement the *Checker* and use the ***Strategy*** design pattern to write two checking operations: the first checker validates a given section if the title starts with a capital letter and the content contains more than 2 sentences; the second validates a given section if the title contains only one word and the content contains no more than 300 words. **(2p)**
3. Create 2 sections and generate them: the first will be validated with the first type of checker and will pass the validation (will be generated); the second will be validated with the second type of checker and will not pass the validation (will not be generated). **(1p)**

6. Practical Examination

Below you will find a problem statement similar to what you can expect to receive during the practical examination. The problem statement will generally follow the requirements set out during this course, will require a graphical user interface (using JavaFX) and writing specifications, tests (if required) and the implementation of layered architecture.

Observations:

1. Solving the following problem statement completely should be possible for you in a time span of 60 minutes (Mathematics Computer Science), 90 minutes (Artificial Intelligence).

2. You are encouraged to bring your own laptop to the exam. You are free to use your preferred IDE. Make sure your IDE is set up correctly and it works!

3. You can start from the provided workspace and you are allowed to use any materials and resources, except for AI generated source code. But you must work **individually**!

6.1. Problem statement (Mathematics Computer Science)

A bus company requires an automatic system for booking. All available routes are contained in a relational database table. A **Route** has *a source city*, *a destination city*, *the departure and arrival times*, *the total available number of seats* and *the price for one ticket*. Write a Java application, with a graphical user interface (using JavaFX), which allows the following:

1. Show all routes in a list **(1p)**, sorted by departure city and departure time. Use Java streams for this sorting **(1p)**. If you do not use Java streams, the maximum score is **0.5p**.
2. The client has a window allowing them to choose the source city from a combo box. The destination city combo box will be automatically updated to contain only the cities reachable from the selected source city **(2p).**
3. After the client chooses the source and destination cities, a new list will show all available routes, for the selected cities, with the following information: source city, destination city, departure and arrival times, duration and ticket price **(1.5p).**
4. A client can select the desired route and book a given number of tickets. The available number of seats for that route must be updated **(1p)**. The total price will be shown in the client's window **(2p).** If there are not sufficient tickets available, the application will show a message in a new window **(0.5p)**.

Bonus:

Show 2 client windows, which both update as the clients make bookings. Use the Observer design pattern. **(1p)**

1p of

6.2. Problem statement (Artificial Intelligence)

Write a Java application, with a graphical user interface (using JavaFX), which simulates the process of package delivering by couriers, as follows:

1. The information about all registered couriers is in a table in a relational database. Each **Courier** has a *name* (string), a *list of assigned streets* and a *zone*, given as a circle (coordinates for centre and a radius).
2. Another table in the same database contains information about packages. Each **Package** has a *recipient* (string), an *address – street and number*, the *location* (2 coordinates), and a *delivery status* (bool).
3. When the application is launched, a new window is created for each courier, having as title the courier's name. The zone for the courier is also shown **(0.25p)**. The window will show in a list only the undelivered packages (with all their information) that are assigned to the courier: the street on the package is in the list of streets of the courier or the location of the package is within the courier's zone. Use Java streams for this operation. **(1.5p)**.
4. The courier's window includes a button "Optimise route", which will show the order of delivery of undelivered packages, in a list, starting from the courier's centre point. Use any algorithm for this calculation. **(0.75p)**
5. Another "courier company" window shows all packages, with all their information **(0.5p)**. Using this window, any company employee can add a new package, by inputting its information. When adding a new package the delivery status is *false*. **(0.75p)**
6. In the courier window, a combo box will be populated with all streets and will allow the couriers to see only the packages for the selected street **(1.5p)**.
7. Couriers can deliver packages by selecting a package and pressing a "Deliver" button. The package's status delivery changes to true and the package disappears from the courier's window **(0.75p)**.
8. A new "map" window will show all undelivered packages, with their location **(0.5p)**.
9. When any modification is made by any courier or by a company employee, everyone will see it, automatically. The "map" window will also be updated automatically. **Use the Observer design pattern. (2p)**
10. When the application closes, the packages table will be updated. (**0.5p**)

1p of

6.3. Advice for the practical examination

1. Implement a problem similar to the one in the example (Section **Problem statement**). Time yourself while solving it, make sure you can implement at least some of the functionalities in the allotted time. This way you can detect where your difficulties are and you can improve yourself.
2. Build your application incrementally: one step at a time, **run frequently**.
3. Only add one function in one step, so that you can easily revert to a functional version, in case something doesn't work.
4. **Do not ignore the errors**, solve them before continuing with writing source code. **Read the error text!**
5. Do not ignore warnings, sometimes these can indicate errors in the program.

6. **Do not implement functionalities that are not required**. By doing this, you might waste valuable time and **the source code will not be graded, only the functional requirements**!
7. If there are issues that you cannot solve, try finding alternative implementations such that you can still test your code.