Labs - Contemporary Software Development
Spring 2022

Contents

1 Formalia

Labs II and III are mandatory and must be successfully completed in order to secure the corresponding 4 credits in this course.

Before attempting Labs II and III it is highly recommended that you both watch the pre-recorded lessons on basic object-oriented programming and successfully complete Lab I. Both of these activities are optional but if you lack a solid understanding of basic object-oriented programming then Labs II- and III will be quite difficult and it is therefore recommended that you start with Lab I. The pre-recorded lessons can be found in a playlist on YouTube.

If you are new to programming, you are also recommended to use the application Microsoft Visual Studio and to engage with the content on getting started with Visual Studio published by Microsoft.

Before attempting Lab III it is recommended that you watch through the pre-recorded lecture series on object-oriented design patterns. The design patterns series can be found on YouTube.

1.1 Supervision sessions

To support your lab work we offer a number of optional tutoring sessions with the teaching assistants (TAs) on this course. The tutoring sessions are not planned lessons. This means that you are expected to bring questions that you wish to discuss with your TA to each session. Do not expect the TA to solve the lab problems for you. Their job is to suggest general strategies that will help you approach your problem.

The lab assignments, and thus the supervision, is done in lab groups of 2 students. To participate in the supervision, you must register for a group and then sign-up for a each session you wish to participate in. To register for a lab group, you must go to Studium and pick a group (which are found under the menu option "Persons"). Do note that choosing a group will restrict you to a specific supervisor. You are allowed to change groups freely until February 3rd, but after that point you will need to contact one of the course teachers.

The tutoring sessions are visible in the course schedule but to attend you must, in advance, book a time-slot via Studium. Time-slots for the next tutoring session are made available for booking via Studium in conjunction to the previous tutoring session.

Between the tutoring sessions, we strongly encourage you to participate in and start discussions on the forum on Studium. We also encourage you to form study groups with your fellow students.

1.2 Grading procedure

- Your submissions are marked as either pass (G) or fail (U).
- Only briefly written comments will be offered for each submission. To receive more extensive feedback on your progress, you should attend the optional tutoring sessions.
- There are two deadlines per lab. These are listed in Table 1.
- Submissions are graded after each deadline has passed.
- Grading at other times will only be granted in cases of e.g. severe illness, death in the family, and similar. Please contact a teaching assistant or the head teacher in such cases as soon as possible.
- If you fail to meet the criteria for this module during spring 2022, you must resubmit the next time the course is run. It is your responsibility to notify the head teacher for that course instance, in due time, that you wish to resubmit so that the teacher can inform you about any changes in the grading criteria and how to submit. You do not need to re-register to the course in order to submit. Please contact the student expedition of the department of informatics and media (info@im.uu.se) if you have any further questions about this and if you need to find out who the head teacher is for a course.

Lab	**Deadline 1**	**Deadline 2**
Lab I	n/a	n/a
Lab II	22/2 (23:59)	23/3 (23:59)
Lab III	23/3 (23:59)	20/4 (23:59)

Table 1 – Lab deadlines.

1.3 Submission

- Labs are submitted to Studium.
- Code must be submitted as a compressed archive in either .zip, .rar, or .7z format. Make sure to submit your full code base. For C#, this means that you should compress either the folder that contains your `.sln` file or the folder that contains your `.csproj` file - depending on whether you are working with a full solution or only a single project or if you are working with a single project and no solution.
- Lab reports must be submitted in a .pdf format.

1.4 Cheating and plagiarism

The work you submit must original work. Suspected cases of cheating and/or plagiarism are escalated and handled in accordance with Uppsala University's Guidelines on Cheating and Plagiarism.

Cheating and plagiarism is found in almost every course. Every year a few of these cases lead to suspensions.

Do not cheat.

2 Lab I: Quickstart

Lab I is *optional*, but if you are struggling to complete Lab II then you are strongly advised to attempt Lab I and use the tutoring sessions to discuss any thoughts you may have.

When you have both completed Lab I and actively watched the pre-recorded lab lectures, you are expected to be familiar with the basic theoretical components of object-oriented programming (e.g. encapsulation, message passing, object composition, subtyping, and inheritance, etc.) and the terminology we use in object-oriented languages (e.g. classes, objects, instance members, etc.).

2.1 Instructions

Download, analyze, and run the console/terminal/command line application found here and complete or answer the following challenges or questions.

1. **What is a variable?**
2. Find an example of variable declaration.
3. Find an example of variable initialization.
4. Find an example of variable assignment.
5. **What is a method?**
6. Find an example of a method signature.
7. Find an example of a method parameter.
8. Find an example of a method return type.
9. **What is a class?**
10. Find an example of a class.
11. Find an example of instantiation.
12. **What is an instance member?**
13. Find an example of an instance variable.
14. Find an example of an instance method.
15. **What is a static member?**
16. Find an example of a static variable.

17. Find an example of a static method.
18. **What is object composition?**
19. Find an example of object composition.
20. **What is overloading/subtype polymorphism?**
21. Find an example of interface implementation.
22. Find an example of a super class.
23. Find an example of a subclass.
24. Find an example of inheritance.
25. Find an example of method overloading.
26. Find an example of a polymorphic call to an overloaded method.
27. **hat is type casting?**
28. Find an example of upcasting.
29. Find an example of downcasting.
30. **What is an access modifier?**
31. Find an example of an access modifier.
32. What is the difference between the access modifiers public, private, and protected?
33. What are some other forms of polymorphism?
34. Find an example of method overriding.
35. Find an example of parametric polymorphism (generics).
36. **What is modularity?**
37. What does the application do?
38. What does the application print?
39. What are some other programs that we could build by composing the classes of this program in different ways?

3 Lab II: Object-oriented programming

This lab is **mandatory** and is more difficult than Lab I as its purpose is to establish that you can solve basic problems using object-oriented techniques without being guided step-by-step.

3.1 Deliverables

1. A software solution.
2. A summarizing lab report.

The software solution must meet the requirements outlined in the problem and requirements sections.

The summarizing lab report must not exceed 500 words and must be submitted in a .pdf format. In the report, you must demonstrate your ability to understand how objects interact with other objects in your established domain and what role your classes serve in solving the problem at hand.

Do note that your report should be written as a personal reflection rather than a summary. This implies that you, e.g., might want to argue the utilization of polymorphism over conditionals in your application rather than to simply describe that you have used polymorphism.

• Classes, objects, and messages	• Instance members (methods and variables)
• Object composition	• Static members (methods and variables)
• Overloading polymorphism (subtyping)	• Access modifiers
• Overriding polymorphism	• Runtime errors (exceptions)
• Parametric polymorphism	• Compile time errors
• Downcasting and upcasting	• Pre-conditions, post-conditions, invariants

Table 2 – Some terminology used in object-oriented programming.

3.2 Problem description

Imagine that we are drawing a set of (potentially overlapping) squares and circles on a piece of (infinitely large) paper. Imagine then that we take a pen, close our eyes, and draw a dot at a random location on the paper. When we open our eyes again, the point will be either be "inside" or "outside" of each shape. Let us refer to this as a *hit* or a *miss*, respectively.

Let `p` represent any point in this two-dimensional coordinate system (i.e. the paper). Let `h(p)` and `m(p)` be two functions that return the set of shapes that are hit and missed, respectively, by point `p`. The score of point `p` can then be computed as:

$$pointScore(p) = \sum_{x \epsilon h(p)} shapeScore(x) - 0.25 \sum_{y \epsilon m(p)} shapeScore(y) \tag{1}$$

Meaning that we take the difference between the sum of the shape score for all shapes that were hit and one fourth ($\frac{1}{4}$) of the sum of the shape score for all shapes that were missed. The shape score function is defined as:

$$shapeScore(s) = \frac{typePoints(s) * instancePoints(s)}{area(s)} \tag{2}$$

where `area(s)` computes the area of shape `s`, `typePoints` returns 2 if `s` is a circle and 1 if it is a square, and `instancePoints(s)` returns a number representing the points for shape `s`, which has been provided as part of the input.

Input is provided as two command line arguments. The first argument is a point in a two-dimensional coordinate system, while the second argument is a list of shapes along with their positions, sizes and points.

The first input argument is specified as a single point (i.e., a coordinate) in a coordinate system with the form `(x,y)`. Whitespace is insignificant meaning that spaces in the input argument should be ignored, i.e., `( x ,   y)` should be interpreted in the same way as `(x,y)`.

`x` represents the value of the X-coordinate and `y` represents the value of the Y-coordinate. Thus, both `x` and `y` are represented with the datatype `integer`.

An example of the first input argument may therefore look something like:

```
(1,0)
```

The second input argument represents the shapes on the "virtual" game board and their information. This input argument arrives in a CSV format, where columns are delimited by comma `(,)` and rows by semi-colon `(;)`.

Note that the first row contains headers and that the order of columns might vary between inputs. The individual columns of this argument is described in Table 3.

Column	Data type	Description
SHAPE	CIRCLE | SQUARE	Shape type.
X	Integer	X-coordinate of the shape.
Y	Integer	Y-coordinate of the shape.
LENGTH	Integer	Circumference of the shape.
POINTS	Integer	Instance points of the shape.

Table 3 – Input description for argument 2.

In the example below, we receive four shapes where the first two are *circles* and the last two are *squares*.

```
SHAPE, X, Y, LENGTH , POINTS;
CIRCLE, 3, 1, 13, 100;
CIRCLE, 1, -1, 15, 200;
SQUARE, -1, 0, 20, 300;
SQUARE, -3, 2, 8, 400;
```

Note that since whitespace must be insignificant in all input arguments, the above input example is equivalent to:

```
SHAPE,X,Y,LENGTH,POINTS;CIRCLE,3,1,13,100;CIRCLE,1,-1,15,200; SQUARE
, -1 ,0 ,20 ,300; SQUARE , -3 ,2 ,8 ,400;
```

Additionally, keep in mind that the order of the columns is insignificant and that the values in the input argument are always based on the order listed in the first "row". For example, the above list of shapes can also be written as:

```
SHAPE,Y,LENGTH,X,POINTS;CIRCLE,1,13,3,100;CIRCLE,-1,15,1,200; SQUARE
, 0 ,20 ,-1 ,300; SQUARE , 2 ,8 ,-3 ,400;
```

Here, we have changed the order of the headers and - more specifically - the order in which the X-coordinate is located for each shape.

Finally, a complete input for a single round of the game may look something like:

```
"(1,0)"
"SHAPE,X,Y,LENGTH,POINTS;CIRCLE,3,1,13,100;CIRCLE,1,-1,15,200; SQUARE
, -1 ,0 ,20 ,300; SQUARE , -3 ,2 ,8 ,400;"
```

Do note that we enter both inputs at the same time and that all inputs will initially be treated as `strings`.

3.2.1 Visualization of the problem

This lab does not require any form of visualization (meaning you are **not** expected to develop a UI) but Figure 1 shows how we could visualize the example input above. Given that the first input argument is `(1,0)`, the point whose score we seek to compute is, in Figure 1, marked with a black dot. Note that the point resides inside both the blue square and the green circle but not the purple square nor the red circle.

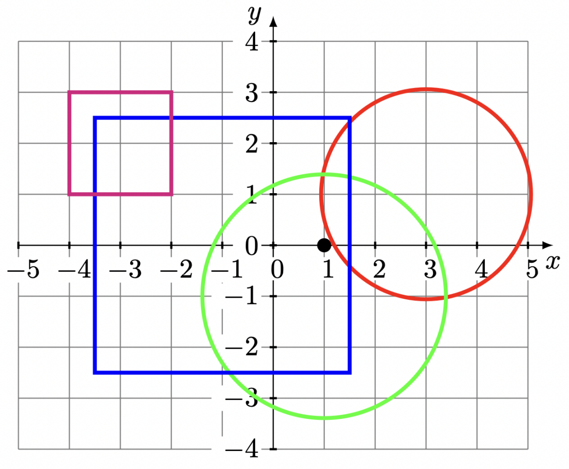


Fig. 1 – Visualization of the example input in Lab II.

To compute the score of the point we would do:

$$pointScore(1,0) = \frac{2*200}{\frac{15^2}{4\pi}} + \frac{1*300}{(\frac{20}{4})^2} - 0.25(\frac{2*200}{\frac{13^2}{4\pi}} + \frac{1*400}{(\frac{8}{4})^2} \approx 5.6224 \quad (3)$$

Your command line application should read the two command line arguments and respond with the point score for the point provided in the first argument, rounded to the nearest integer. For example, when run with the two arguments provided in the example above, we should print 6 in the terminal.

When using a command line application, you enter input arguments via a terminal (you can find more information about this process in the guiding documents located on Studium). In this particular case, running the input might look something like this:

```
dotnet run "(1,0)"
"SHAPE,X,Y,LENGTH,POINTS;CIRCLE,3,1,13,100;CIRCLE,1,-1,15,200; SQUARE
, -1 ,0 ,20 ,300; SQUARE , -3 ,2 ,8 ,400;"
```

Which would the print the result, also in said terminal, 6 (i.e., 5.6224 rounded to the nearest integer).

3.3 Requirements

While most of the functional requirements are specified in the description, the program must also adhere to the following requirements:

- The program must be a console (i.e. command line/terminal) application written in C#.
- The program uses no static classes/methods/variables (except for the `Main`-method).
- The program must be fault-tolerant and gracefully notify the user in case of syntactically invalid input, missing arguments, and other runtime errors that might occur.
- The program must contain at least 4 *meaningful* classes, in addition to the `Program`-class.

4 Lab III: Object-oriented design

This third lab is **mandatory** and also more complex than the second. Not because the functional requirements are necessarily more challenging, but because the non-functional requirements and the lab report are and is. In this lab you are expected to be able to not only independently produce software, but also motivate and critically reflect upon design decisions in relation to maintainability.

4.1 Deliverables

- A software solution.
- A reflective lab report.

The software solution must meet the requirements outlined in the problem and requirements sections.

The reflective lab report must be no longer than 1,500 words and must be submitted in a .pdf format. In your report, you must demonstrate your ability to use object-oriented terminology (see Tables 2 and 4) to motivate and critically reflect upon your design decisions and your chosen design patterns in relation to software maintainability. This means that you must:

1. argue for your chosen design patterns. E.g., by motivating why these specific patterns were chosen rather than other patterns,
2. what purpose these patterns provide to your application and
3. how these patterns are implemented

• Encapsulation and information hiding	• Open/closed principle
• Coupling and cohesion	• Liskov substitution principle
• Replace conditionals with polymorphism	• Interface segregation principle
• Primitive obsession and Message obsession	• Dependency inversion/injection
• Making impossible states impossible	
• Single responsibility principle	

Table 4 – Some terminology and design principles used in object-oriented design.

Maintainability is, by ISO/IEC 25010:2011 (ISO, 2011), defined as the *"degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers"*. Further dimensions of maintainability are provided in the excerpt given in Table 5.

Modularity	Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
Reusability	Degree to which an asset can be used in more than one system, or in building other assets.
Analysability	Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
Modifiability	Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
Testability	Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

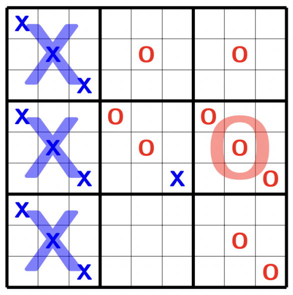
Table 5 – Excerpt on the definition of software maintainability (ISO, 2011).

4.2 Problem description

An interesting variation of the classic game tic-tac-toe is known as super, strategic, or meta tic-tac-toe. In this "upscaled" version, each tic-tac-toe board serves as a square in a larger tic-tac-toe board. When a player, say X, wins a full "small" board, this player gets to play the corresponding square in the "larger" board. Consider Figure 2a and let us refer to the large 3x3 grid as a "large" board and each of the 9 individual 3x3 grids as the "small" boards.

In the game represented in Figure 2a, we observe that X has won three small boards and thereby gained three X:es in the large board. The second player (O) have, however, merely won a single of the small boards and have hence only gained a single O in the large board.

Since X has won three small boards that together constitute three-in-a-row in the large board, X is the winner of this match.



(a) Visualization.

```
NW.CC, NC.CC,
NW.NW, NE.CC,
NW.SE, CE.CC,
CW.CC, SE.CC,
CW.NW, CC.CC,
CW.SE, CC.NW,
CC.SE, CE.NW,
SW.CC, CE.SE,
SW.NW, SE.SE,
SW.SE
```

(b) Input.

Fig. 2 – Super tic-tac-toe example.

Sometimes this game is played with a rule that dictates which small board a player is allowed to play in as a consequence of the square in a small board that the last player played. In this exercise we ignore such complications and assume that players are allowed to play whatever square they want to play so long as each square is only played once.

Your program must read input data from a match between X and O and return information concerning how the match has played out.

All input is provided as a single command line argument. This argument is a comma-separated (,) list of moves where players take turn to make their moves. Assuming that X makes the first move, all odd numbered moves belong to X while all even belong to O.

The exception from this rule is if a player tries to make an invalid move. If this happens, we simply assume that the next move in the list of moves belongs to the player in question since the first move was rejected. I.e., if player X attempts and invalid move, the next move from the move-list still belongs to player X.

A move is only ever rejected if a player tries to either (1) play a square (in a small board) that has already been played (i.e., the square is already occupied) or (2) if a player attempts to make a move in a small board which already has a winner.

4.2.1 Input example

Let us consider a complete input example before moving on. An example input string is provided in Figure 2b and a visualization based on that input is provided in Figure 2a.

As you may have noticed, we use abbreviations to refer to different positions on the board. The abbreviations are based on the words: north (N), west (W), east (E), south (S), and center (C). It is perhaps best explained by means of the visualizations in Figure 3.

NW	**NC**	**NE**
CW	**CC**	**CE**
SW	**SC**	**SE**

(a) depth = 1.

NW. NW	NW. NC	NW. NE	NC. NW	NC. NC	NC. NE	NE. NW	NE. NC	NE. NE
NW. CW	NW. CC	NW. CE	NC. CW	NC. CC	NC. CE	NE. CW	NE. CC	NE. CE
NW. SW	NW. SC	NW. SE	NC. SW	NC. SC	NC. SE	NE. SW	NE. SC	NE. SE
CW. NW	CW. NC	CW. NE	CC. NW	CC. NC	CC. NE	CE. NW	CE. NC	CE. NE
CW. CW	CW. CC	CW. CE	CC. CW	CC. CC	CC. CE	CE. CW	CE. CC	CE. CE
CW. SW	CW. SC	CW. SE	CC. SW	CC. SC	CC. SE	CE. SW	CE. SC	CE. SE
SW. NW	SW. NC	SW. NE	SC. NW	SC. NC	SC. NE	SE. NW	SE. NC	SE. NE
SW. CW	SW. CC	SW. CE	SC. CW	SC. CC	SC. CE	SE. CW	SE. CC	SE. CE
SW. SW	SW. SC	SW. SE	SC. SW	SC. SC	SC. SE	SE. SW	SE. SC	SE. SE

(b) depth = 2.

Fig. 3 – Coordinate system in super tic-tac-toe.

Note that we are using the "dot-syntax" to "dig down" into smaller boards. The coordinate `NW.CC`, for example, refers to the middle square ("center center") in the top left small board ("north west"). While this syntax can be used to specify games at an arbitrary depth, we observe one and two layers in Figures 3a and 3b respectively.

No matter how many layers we employ, players always make their moves in the "inner-most"/"smallest" squares and the winner is always determined using the "outermost"/"largest" squares.

Your program must be able to handle at least a depth of 2 and determine depth on the basis of the first move. Note that the depth of all moves must be the same. For example, we cannot have `"NW.CC"` and `"NW.CC.SE"` in the same input argument as they represent two different depths.

Whitespace must be insignificant in the input argument. As such, the line breaks and spaces used in Figure 2a are only added to improve readability of this document.

4.3 Outputs

When given some correctly formatted input, your program must return three lines of output, as per the following specification:

Line 1: A comma-separated `(,)` list of the winning "large" squares in the order in which they were played. Given the input in Figure 2b our program would print:

```
NW, CW, SW
```

Line 2: A comma-separated `(,)` list of the winning "small" squares in the order in which they were played. In the example of Figure 2b this is fairly straight forward as all the moves, of player one, except one are winning moves. The output should in this case be:

```
NW.CC, NW.NW, NW.SE, CW.CC, CW.NW, CW.SE, SW.CC, SW.NW, SW.SE
```

Line 3: Two values, separated by a comma `(,)` where the first (i.e. left) value denotes how many games player X has won while the second (i.e. right) value denotes that of player O. Wins are counted at every layer and layers are separated with dots. In the example of Figure 2b we have two layers and should thus print two numbers per player. The first player, `(X)`, has won one (of one) of the "big" boards and three (of nine) of the "small" boards. The second player, `(O)`, has won zero (of one) "big" boards but one (of nine) "small" boards. Consequently we should print:

```
1.3, 0.1
```

4.4 Requirements

While most of the functional requirements are specified in the description, the program must also adhere to the following requirements:

- The program must be a console (i.e. command line/terminal) application written in C#.
- The program must be fault-tolerant and gracefully notify the user in case of syntactically invalid input, missing arguments, and other runtime errors that might occur.
- The program must contain at least 6 *meaningful* classes.
- The program must not contain any static classes/methods/variables (except for the Main-method).

- The program must not use downcasting or type checking.
- The program must prefer polymorphism over conditionals.
- The program must adhere to the SOLID principles.
- The program must exhibit meaningful use of at least **two** design patterns. You may freely choose which design patterns to apply from the list in Table 5. Design patterns not listed in Table 5 will not count towards the total.
- The program must contain at least one automated test that ensures that the input example (see Figure 2) yields the expected output.

• Abstract factory	• Interpreter
• Bridge	• Mediator
• Builder	• Memento
• Chain of responsibility	• Observer
• Command	• Prototype
• Composite	• State
• Decorator	• Template method
• Factory method	• Visitor

Table 6 – Allowed design patterns.