**CONCORDIA UNIVERSITY**
**DEPARTMENT OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING**

**SOEN 6441: ADVANCED PROGRAMMING PRACTICES: SECTION V**
**WINTER 2023**

# PROJECT DESCRIPTION

## 1. INTRODUCTION

For many, a cold beverage in a glass is among the means for survival during summer. The main purpose of a coaster is to absorb condensation dripping along the glass, and thereby protect the surface of a table or any other surface used for placing the glass. The coasters are also of interest to **tegestologists**.

For the sake of reference, the project as well as its product is called **CHEERS**.

## 2. PROBLEM

Let there be two circular coasters of equal area (and negligible height). The purpose of **CHEERS** is to find how far the two coasters need to be moved on top of each other such that the area of the overlapping region is half the area of any one of the coasters, as illustrated in Figure 1.

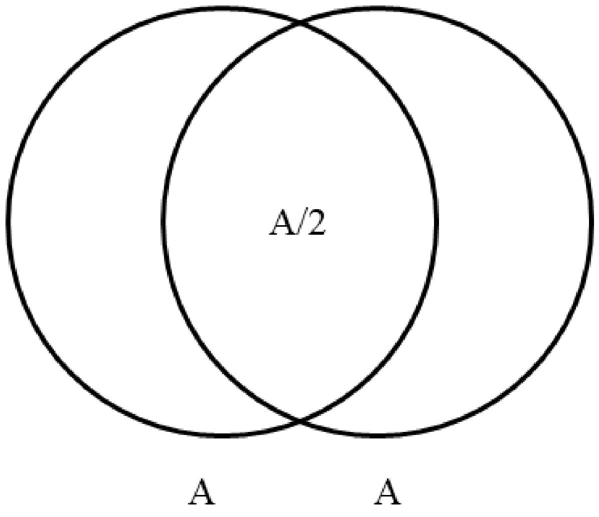

Figure 1.

## 3. SOLUTION

The segment $X_1X_2$ on the $x$-axis is an indicator of the degree of overlap between the two coasters, as illustrated in Figure 2.

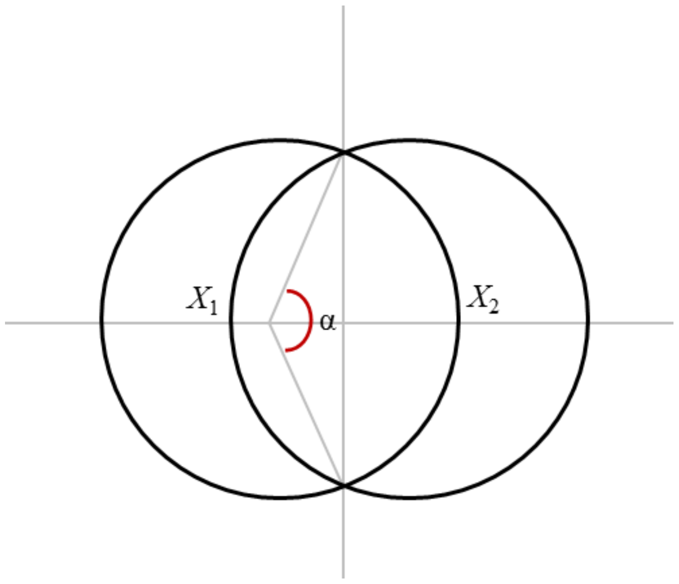

Figure 2.

Let $R$ be the radius of any one of the coasters, and let α be the angle with vertex at the center of the left circle, as shown in Figure 2.

Then the length $l$ of the segment $X_1X_2$ is given by the equation

$$l = 2R(1 - \cos(\alpha/2)),$$

and α is given by the equation

$$\alpha - \sin(\alpha) = \pi/2.$$

## 4. CHEERS

The purpose is to compute $l$. (This, in turn, requires computing α.)

The work on **CHEERS** has been divided into an **interrelated collection of problems** to be solved. Each problem has a note associated with it, meant to serve as a guide for scoping, understanding, and/or solving the problem.

## 5. DELIVERABLE 1 (D1)

This is about implementation, at a local level.

Let **T** be an arbitrary project team, let **P** be the program developed for **CHEERS** by **T**, and **S** be the source code of **P**.

### PROBLEM 1. [50 MARKS]

**T must** provide an outline of the solution, including the object-oriented design corresponding to **P**.

**T must** construct a **CRC card model**.

**NOTE**

For each CRC card, the following must be described: (a) role, (b) responsibilities and the rationale for the responsibilities, and (c) collaborations and reasons for collaborating.

### PROBLEM 2. [50 MARKS]

**T must** rationalize the selection of each algorithm deployed in **S**, and document the algorithms using **pseudocode** (as well as natural language, if necessary).

**NOTE**

The pseudocode should not be too close to the implementation.

### PROBLEM 3. [90 MARKS]

(a) This is **Incarnation 1**.

**S must** be written from **scratch**. **S must not** make use any native support in programming languages for trigonometric functions, such as sin($x$) and cos($x$), and for the mathematical constants, such as $\pi$. **S must not** make use any application programming interfaces or library functions, other than for input, output, and basic arithmetic.

**T must not** use any source code from any other courses or any place on the Web.

**NOTE**

The following applies to (a) only.

**T must** provide the steps taken towards making **S** to be **modifiable, readable, reusable, testable, and understandable**.

**T must** provide the steps taken towards making the corresponding **P** to be **general, robust, and usable**.

**T must** provide a sample output, say, for different values of *R*. The **output** of **P** must be in structured plain text.

(b) This is **Incarnation 2**.

**S must** be written in a manner that seeks as many **opportunities for reuse** as possible. **S** could, for example, make use of application programming interfaces or library functions, available **natively or otherwise**.

**T must not** use any source code from any other courses.

**NOTE**

The following applies to (b) only.

**T must** provide the steps taken towards making **S** to be **readable, modifiable, testable, and understandable**.

**T must** provide the steps taken towards making the corresponding **P** to be **general, robust, and usable**.

**T must** provide a sample output, say, for different values of *R*. The **output** of **P** must be expressed in XML. This output must be valid with respect to some XML DTD.

**NOTE**

The following apply to both (a) and (b).

**S must** be written in **Python**. **T must** select the same version of Python (2.x or 3.y) for both (a) and (b).

**S must not** depend on any particular development environment (such as a specific operating system or an IDE). (**T must** prove that **S** is independent of any IDE.)

**S must** be placed on a **distributed version control system**. **S must** evolve **iteratively, incrementally, and regularly**. (**T** must **at least twice a week** commit **non-trivially** to a distributed version control system, and make his or her account accessible to the teaching assistants.) **T** is **encouraged** to use **Semantic Versioning** for versioning of artifacts.

**T must** use a debugger. (**T** must prove that a debugger was used.)

**S must** follow a **style of programming** recommended by an authoritative source. (For Python, one such authoritative source is **PEP 8**.) The actual style of programming must aim to be **consistent** across **T**. **T** must provide a pointer to the style guidelines used.

**S must** be **documented appropriately** using a **'standard' documentation system**. (For Python, such a documentation system is Pydoc.) It is recommended that the internal documentation and the source code **evolve synchronously**.

**S must not** violate established **principles of programming**. (For example, such principles for **object-oriented programming** are those related to abstraction, encapsulation, inheritance, and/or polymorphism.)

**S must not** declare π as a named constant. (It must compute the value of π.)

**S must** have proper support for **handling exceptions**.

**S must** have proper **error messages**.

**T must** provide a listing of **S**.

**T must** provide a description of instructions for processing **S** (by compiling or interpreting it, as the case may be).

**T must** provide information related to **any (reasonable) assumptions made by S, references for supporting technical arguments, claims, and any non-original work** (that is, any work external to **T**), and so on. A comprehensive collection of resources on citing and referencing is available[1]. For example, ACM, APA, and IEEE provide standard formats for citing and referencing. It is important not to make claims that cannot be substantiated, and not to copy others' work verbatim regardless of whether it is cited. A **work that is copied in any manner (syntactically, semantically, or pragmatically) does not earn any credit**.

[1] URL: http://library.concordia.ca/help/howto/citations.html .

## 6. DELIVERABLE 2 (D2)

This is about testing, at a global level. Indeed, the testing processes herein turn the class into a rudimentary **social network**.

### PROBLEM 4. [40 MARKS]

This is about **static testing**.

**S must** be **reviewed** (inspected) systematically against the "best practices" and style guidelines it is supposed to conform to.

**NOTE**

The **static testing process** must **non-reciprocally rotate** across the class.

Let $T_1$, …, $T_n$ be the teams in the class. To make the process predictable, $T_1$ will review **S** $T_2$, $T_2$ will review **S** by $T_3$, …, $T_n$ will review **S** by $T_1$. (This is always possible if $n > 2$.)

Let $T_\alpha$ and $T_\beta$ be two arbitrary teams in the class. Then, a static testing of **P** by $T_\alpha$ can be carried out by $T_\beta$, but **not conversely**, that is, a static testing of **P** by $T_\beta$ cannot be carried out by $T_\alpha$.

$T_\alpha$ **must** make available **all** its incarnations to $T_\beta$.

$T_\beta$ **must** provide a record of the outcome of review.

### PROBLEM 5. [40 MARKS]

This is about **dynamic testing**.

**P must** be tested systematically. There must be a list of test cases. There must be a description of each test case. A test case must clearly outline the input and expected output.

**NOTE**

The use of **test frameworks**, such as those for **unit testing and acceptance testing** (available at `http://opensourcetesting.org/`), is encouraged, but **not required**.

The **dynamic testing process** must **non-reciprocally rotate** across the class.

Let $T_1$, …, $T_n$ be the teams in the class. To make the process predictable, $T_1$ will test **P** by $T_2$, $T_2$ will test **P** by $T_3$, …, $T_n$ will test **P** by $T_1$. (This is always possible if $n > 2$.)

Let $T_\alpha$ and $T_\beta$ be two arbitrary teams in the class. Then, a dynamic testing of **P** by $T_\alpha$ can be carried out by $T_\beta$, but **not conversely**, that is, a dynamic testing of **P** by $T_\beta$ cannot be carried out by $T_\alpha$.

$T_\alpha$ **must** make available **all** its incarnations to $T_\beta$.

$T_\alpha$ **must** provide a collection of test cases.

$T_\beta$ **must** provide a record of the outcome of testing.

## 7. DELIVERABLE 3 (D3)

### PROBLEM 6. [30 MARKS]

**T** must give a presentation that (a) using sample data, **demonstrates and compares both the incarnations** of **P**, and (b) highlights **lessons learned** from working together, reviewing, and testing efforts, and (c) points out **limitations** of any tools used.

**NOTE**

The presentation should aim to be memorable.

Each member of every team must create his or her own set of slides. This set could be based on a discussion with the others in the team and overlap with that of the others.