Programmering og Problemløsning
Datalogisk Institut, Københavns Universitet
Uge(r)seddel 11 - group opgave

Jon Sporring and Christina Lioma

18. december – 10. januar.
Afleveringsfrist: onsdag d. 10. januar kl. 22:00

I denne periode skal I arbejde i grupper. Formålet er at arbejde med:

- Inheritance
- UML diagrams

Opgaverne for denne uge er delt i øve- og afleveringsopgaver.

Øve-opgaverne er:

11ø.0 Write a `Person` class with data attributes for a person's name, address, and telephone number. Next, write a class named `Customer` that is a subclass of the `Person` class. The `Customer` class should have a data attribute for a customer number and a Boolean data attribute indicating whether the customer wishes to be on a mailing list. Demonstrate an instance of the `Customer` class in a simple program.

11ø.1 Write an `Employee` class that keeps data attributes for the following pieces of information:

- Employee name
- Employee number

Next, write a class named `ProductionWorker` that is a subclass of the `Employee` class. The `ProductionWorker` class should keep data attributes for the following information:

- Shift number (an integer, such as 1 or 2)
- Hourly pay rate

The workday is divided into two shifts: day and night. The shift attribute will hold an integer value representing the shift that the employee works. The day shift is shift 1 and the night shift is shift 2. Write the appropriate methods for each class.

Once you have written the classes, write a program that creates an object of the `ProductionWorker` class and prompts the user to enter data for each of the object's data attributes. Store the data in the object and then use the object's methods to retrieve it and display it on the screen.

11ø.2 Extend the previous exercise as follows: Let a shift supervisor be a salaried employee who supervises a shift. In addition to salary, the shift supervisor earns a yearly bonus when his or her shift meets production goals. Write a `ShiftSupervisor` class that is a subclass of the `Employee` class you created in the previous exercise. The `ShiftSupervisor` class should keep a data attribute for the annual salary and a data attribute for the annual production bonus that a shift supervisor has earned. Demonstrate the class by writing a program that uses a `ShiftSupervisor` object.

11ø.3 **(Extra difficult).** Considering that production during night shifts is reduced by 5% compared to production during day shifts, and that the hourly pay rate during night shifts is double the hourly pay rate during day shifts, compute the best possible worker & shift allocation over the period of 12 months. You need to think how to measure productivity and salary cost, and then find their best tradeoff in the period of 12 months.

11ø.4 Cheetahs, antelopes and wildebeests are among the world's fastest mammals. This exercise asks you to simulate a race between them. You are not asked to simulate their movement on some plane, but only some of the conditions that affect their speed when running a certain distance.

Produce a UML diagram representing the following.

Your base class is called `Animal` and has these attributes:

- The amount of food needed daily (measured in kilograms)
- The weight of the animal (measured in kilograms)
- The maximum speed of the animal (measured in kilometres per hour)
- The current speed of the animal (measured in kilometres per hour)

The `Animal` class should have a primary constructor that takes two arguments: the animal's weight and the animal's maximum speed. The `Animal` class should also have an additional constructor that takes as input only the animal's maximum speed and generates the animal's weight randomly within the range of 70 - 300 kg. The `Animal` class should have two methods:

- The first method should set the current speed of the animal proportionately to its food intake and maximum speed as follows: if the animal eats 100% of the amount of food it needs daily, the animal's current speed should be its maximum speed; if the animal eats 50% of the amount of food it needs daily, the animal's current speed should be 50% of its maximum speed, and so on.

- The second method should set the amount of food needed daily proportionately to the animal's weight as follows: the animal should eat half its own weight in food every day (if the animal weighs 50 kg, it should eat 25kg of food daily).

Create a subclass `Carnivore` that inherits everything from class `Animal`, and modifies the second method as follows: the animal should eat 8% of its own weight in food every day.

Create a subclass `Herbivore` that inherits everything from class `Animal`, and modifies the second method as follows: the animal should eat 40% of its own weight in food every day.

Create an instance of `Carnivore` called `cheetah` and two instances of `Herbivore` called `antelope, wildebeest`. Set their weight and maximum speed to:

- cheetah: 50kg, 114km/hour
- antelope: 50kg, 95km/hour
- wildebeest: 200kg, 80km/hour

Generate a random percentage between 1 - 100% (inclusive) separately for each instance. This random percentage represents the amount of food the animal eats with respect to the amount of food it needs daily. E.g., if you generate the random percentage 50% for the antelope, this means that the antelope will eat 50% of the amount it should have eaten (as decided by the second method).

For each instance, display the random percentage you generated, how much food each animal consumed, how much food it should have consumed, and how long it took for the animal to cover 10km. Repeat this 3 times (generating different random percentages each time), and declare winner the animal that was fastest on average all three times. If there is a draw, repeat and recompute until there is a clear winner.

Test all methods.

Optional extra: repeat the race without passing as input argument the weight of each animal (i.e. letting the additional constructor generate a different random weight for each instance).

11ø.5 Du skal implementere en udvidelse til Simple Jacksom indeholder en omstrukturering af nogle af klasserne, samt indførelse af en række nye strategier. Du skal simulere nogle Simple Jackspil hvor du afprøver forskellige strategier, for at afgøre hvilken strategi som lader til at være den bedste.

Implementér super-klassen `Player`, og klasserne `Dealer`, `Human` og `AI` som nedarver fra `Player`. `Player` skal indeholde attributter og metoder som implementerer den fælles funktionalitet som alle tre typer "spillere" har, f.eks. en metode som vælger "Hit" eller "Stand".

Implementér super-klassen `Strategy`, samt en klasse for hver af følgende strategier, som alle nedarver fra `Strategy`

(a) Vælg altid "Hit", medmindre summen af egne kort kan være 15 eller over, ellers vælg "Stand".

(b) Vælg altid "Hit", medmindre summen af egne kort kan være 17 eller over, ellers vælg "Stand".

(c) Vælg altid "Hit", medmindre summen af egne kort kan være 19 eller over, ellers vælg "Stand".

(d) Vælg tilfældigt mellem "Hit" og "Stand". Hvis "Hit" er valgt, trækkes et kort og der vælges igen tilfældigt mellem "Hit" os "Stand" osv.

(e) Følg strategi 2. hvis ét af egne kort er et Es, ellers følg strategi 1.

Simulér 3000 spil Simple Jackmed 5 `AI` spillere som følger de 5 ovenstående strategier. Dealer skal følge strategi 2. Konkludér hvilken af strategierne som lader til at være bedst.

Du skal også

- Opdatere dit UML-diagram
- Lave Unittests
- Kommentere ny kode jævnfør kommentarstandarden for F#

11ø.6 Produce a UML diagram for each of the above exercises.

Afleveringsopgaven er:

Sporring, "Learning to program with F#", 2017, Chapter 21.4 describes a simplified version of Chess with only Kings and Rooks, and which we here will call Simple Chess, and which is implemented in 3 files: `chess.fs`, `pieces.fs`, and `chessApp.fsx`. In this assignment you are to work with this implementation, and the assignment consists of the following design and programming tasks:

11g.0 Produce a UML diagram describing the design presented of Simple Chess in the book.

11g.1 The implementation of `availableMoves` for the King is flawed, since the method will list a square as available, even though it can be hit by an opponents piece at next turn. Correct `availableMoves`, such that threatened squares no longer are part of the list of vacant squares. If as a consequence you update the design w.r.t. your UML diagram, then provide an updated UML diagram.

11g.2 Extend the implementation with a class `Player` and two derived classes `Human` and `Computer`. The derived classes must have a method `nextMove`, which returns a legal movement as a codestring or the string "quit". A codestring is a string of the name of two squares separated by a space. E.g., if the white king is placed at a4, and a5 is an available move for the king, then a legal codestring for moving the king to a5 is "a4 a5". The player can be either a human or the computer. If the player is human, then the codestring is obtained by a text dialogue with the user. If the player is the computer, then the codestring must be constructed from a random selection of available move of one of its pieces.

11g.3 Extend the implementation with a class `Game`, which includes a method `run`, and which allows two players to play a game. The class must be instantiated with two player objects either human or computer, and `run` must loop through each turn and ask each player object for their next move, until one of the players quits by typing "quit".

11g.4 Extend `Player` with an artificial intelligence (ai), which instead of picking random moves simulate all possible games at least 2 moves ahead, and returns a movement, which is more likely to lead to a win than random guessing.

11g.5 Make an extended UML diagram showing the final design including all the extending classes.

Krav til opgavebesvarelsen

You must make a program consisting of one or more F# files, that extends Simple Chess as described above, and you must write a small report. The hand-in must consists of the report on pdf-format, one or more fsharp source files, and a single compiled `exe` file, which can be run using `mono` command. The hand-in must also give the list of console commands used to compile the program. Besides the requirements described in the previous section, the program must be documented using the F# documentation standard, and the program must be tested with both a black- and white-box testing. The report must be written in either English or Danish, typeset using LaTeX, and as its main parts include the sections Introduction (Introduktion), Problem analysis and design (Problemanalyse og design), Program description (Programbeskrivelse), Testing (Afprøvning), and Discussion and Conclusion (Diskussion og Konklusion) as shown in `rapport.tex`. As appendix the report must be include a User Guide (Brugervejledning). The report excluding frontpage and appendix should be no larger than 10 pages.

God fornøjelse.