Programmering og Problemløsning
Datalogisk Institut, Københavns Universitet
Arbejdsseddel 11 - gruppeopgave

Jon Sporring

9. december - 3. januar.
Afleveringsfrist: lørdag d. 4. januar kl. 23:59.

Emnerne for denne arbejdsseddel er:

- Objektorienteret design
- nedarvning
- UML diagramer
- statiske værdier (properties) og metoder

Opgaverne er delt i øve- og afleveringsopgaver. I denne periode skal I arbejde i grupper med jeres afleveringsopgaver. Regler for gruppe- og individuelle afleveringsopgaver er beskrevet i "'Noter, links, software m.m."'→"'Generel information om opgaver"'.

Øveopgaver

11ø0 Write a `Person` class with data properties 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 property for a customer number and a Boolean data property indicating whether the customer wishes to be on a mailing list. Demonstrate an instance of the `Customer` class in a simple program.

11ø1 Draw the UML diagram for the following programming structure: A `Person` class has data property for a person's name, address, and telephone number. A `Customer` has data property for a customer number and a Boolean data property indicating whether the customer wishes to be on a mailing list.

11ø2 War is a card game for two players. A simplified version can be described as follows:

> War is a card game for two players using the so-called French-suited deck of cards. The deck is initially divided equally between the two players, which is organized as a stack of cards. A turn is played by each player showing the top of their stack. The player with the highest card wins the hand. Aces are the highest. The won cards are placed at the bottom of the winner's stack. When one player has all the cards, then that player wins the game.

Use the nouns-and-verbs method to identify possible classes and their interactions and write 1-2 lines of description for each.

11ø3 Implement a class `account`, which is a model of a bank account. Each account must have the following properties

- `name`: the owner's name
- `account`: the account number
- `transactions`: the list of transactions

The list of transactions is a list of pairs `(description, balance)`, such that the head is the last transaction made and the present balance. If the list is empty, then the balance is zero. The transaction amount is the difference between the two last transaction balances. To ensure that there are no reoccurring numbers, the bank account class must have a single static field, `lastAccountNumber`, which is shared among all objects, and which contains the number of the last account number. When a new account is created, i.e., when an object of the `account` class is instantiated, the class' `lastAccountNumber` is incremented by one and the new account is given that number. The class must have a class method:

- `lastAccount` which returns the value of the last account created.

Further, each account object must also have the following methods:

- `add` which takes a text description and a transaction amount, and prepends a new transaction pair with the updated balance.
- `balance` which returns the present balance of the account

Make a program, which instantiates 2 objects of the `account` class and which has a set of transactions that demonstrates that the class works as intended.

11ø4 A calendar is a system for organizing meetings and events in time. A description of a calendar is as follows:

> The gregorian calendar consists of dates (day/month/year), with 12 months per year, and with months consisting of 28, 29, 30 or 31 days. The years are counted numerically with Jesus Christus' first year being called 1 BC, followed by 2 BC, etc., and the year prior is called 1 AD, preceded by 2 AD, etc. Thus, this calendar has no year 0.
>
> A user can enter items such as a meeting or an event into a calendar. An item consists of a start date and time, end date and time, and a text-piece. Items can also be whole-day items.

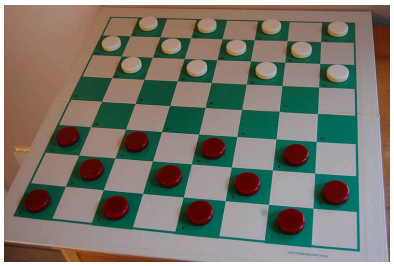

Figure 1: The starting position in Checkers [`https://commons.wikimedia.org/wiki/File:CheckersStandard.jpg`]

Use the nouns-and-verbs method to identify possible classes and their interactions and write 1-2 lines of description for each.

11ø5 Checkers also known as draughts is a ancient board game. A simplified version can be described as follows:

> Checkers is a turn-based strategy game for two players. The game is (typically) played on an 8×8 checkerboard of alternating dark- and light-colored squares. Each player starts with 12 pieces, where player one's pieces are light, and player two's pieces are dark in color, and the initial position of the pieces is shown in Figure 1. Players take turns moving one of their pieces. A player must move a piece if possible, and when one player has no more pieces, then that player has lost the game.
>
> A piece may only move diagonally into an unoccupied adjacent square. If the adjacent square contains an opponent's piece and the square immediately beyond is vacant, then the piece jumps over the opponent's piece and the opponent's piece is removed from the board.

Use the nouns-and-verbs method to identify possible classes and their interactions and write 1-2 lines of description for each.

11ø6 (a) Write an `Employee` class that keeps data properties 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 properties 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 property 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 properties. Store the data in the object and then use the object's methods to retrieve it and display it on the screen.

(b) 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 property for the annual salary and a data property for the annual production bonus that a shift supervisor has earned. Demonstrate the class by writing a program that uses a `ShiftSupervisor` object.

(c) **(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.

11ø7 Make an UML diagram for the following structure:

A `Employee` class that keeps data properties for the following pieces of information:

- Employee name
- Employee number

A subclass `ProductionWorker` that is a subclass of the `Employee` class. The `ProductionWorker` class should keep data properties for the following information:

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

A class `Factory` which has one or more instances of `ProductionWorker` objects.

11ø8 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.

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

- 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).

11ø9 Write a UML diagram for the following:

A class called `Animal` and has the following properties (choose names yourself):

- 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 two methods (choose appropriate names):

- 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).

A subclass `Carnivore` that inherits everything from class `Animal`.

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.

A class called `Game` consisting of one or more instances of `Carnivore` and `Herbivore`.

Afleveringsopgaver

This assignment is about simulating a preditor-prey relationship in a simplified setting.

11g0 You are to simulate owls and mice in a closed environment. The owls are immortal and hunt mice to eat, and the mice run around randomly and multiply (propagate). The overall rules for the simulation are:

(a) The environment must consist of $n \times n$ fields organized as a checkerboard.

(b) Alive animals have a coordinate in the environment, and there can only be one animal per coordinate.

(c) The simulation updates in ticks, and after each tick, all animals perform an action.

(d) The simulation runs for T ticks.

The possible actions are:

(e) A mouse can move to a neighbouring empty field.

(f) A mouse must have a counter, such that after p ticks, the mouse will not move but multiply. The effect is that an offspring is created in an empty neighbouring field. If there is no empty neighbouring field, then the mouse waits a turn.

(g) An owls can move to all neighbouring fields not occopied by another owl. If an owl moves to a field with a mouse, then the mouse is eaten and the mouse is removed from the board.

You are to:

(a) use the object-oriented programming paradigm and include inheritance in your solution.

(b) create a program `simulate.fsx`, which runs the simulation and prints the tick number and the total number of mice after each tick to the textfile `simulation.txt`. The program must accept the parameters n, T, and p at the command-line.

(c) collect the main classes in an implementation file called `preditorPrey.fs`, which `simulate.fsx` links to.

(d) make a white-box test of the implementation file.

(e) find and parameters, where the mice population diminishes to zero quickly, explodes, and is seemingly in balance, and demonstrate this by copying and renaming the textfile `simulation.txt` to the three corresponding files

- `simulationExtinction.txt`,
- `simulationOverpopulation.txt`, and
- `simulationBalance.txt` respectively.

You are also to write a report:

(f) The report must as a minimum include the sections: Introduction, Problem analysis and design, Program description, Testing, Experiments, and Conclusion. Include a User guide and your source code as appendices.

(g) The report must be no longer than 10 pages excluding the appendices.

Afleveringen skal bestå af

- en zip-fil, der hedder `11g_<navn>.zip` (f.eks. `11g_jon.zip`)
- en pdf-fil , der hedder `11g_<navn>.pdf` (f.eks. `11g_jon.pdf`)

Zip-filen `11g_<navn>.zip` skal indeholde en og kun en mappe `11g_<navn>`. I den mappe skal der ligge en `src` mappe og filen `README.txt`. I `src` skal der ligge følgende og kun følgende filer: preditorPrey.fs, simulate.fsx, simulationExtinction.txt, simulationOverpopulation.txt, simulationBalance.txt svarende til hver af delopgaverne. Programmerne skal kunne oversættes med `fsharpc`, og de oversattte filer skal kunne køres med mono. Funktioner skal dokumenteres ifølge dokumentationsstandarden som minimum ved brug af `<summary>`, `<param>` og `<returns>` XML-tagsne. Filen `README.txt` skal ganske kort beskrive, hvordan koden oversættes og køres. Pdf-filen skal indeholde jeres rapport oversat fra LaTeX. Husk at pdf-filen skal uploades ved siden af zip-filen på Absalon.

God fornøjelse.