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

Ken Friis Larsen og Jon Sporring

7. december - 22. december.
Afleveringsfrist: tirsdag d. 22. december kl. 22:00.

Denne arbejdsseddel strækker sig over to uger og indeholder opgaver der fortrinsvis omhandler objektorienteret *design*, med fokus på brug af *nedarvning*. Derudover er der også opgaver der går ud på at bruge UML diagrammer til at dokumentere (og udvikle) jeres designvalg.

Emnerne for denne arbejdsseddel er:

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

Opgaverne er opdelt 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 (in English)

11ø0 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.

11ø1 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 unique customer number and a Boolean data property indicating whether the customer wishes to be on a mailing list. Write a small program, which makes an instance of the `Customer` class.

11ø2 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.

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 AD, followed by 2 AD, etc., and the year prior is called 1 BC, preceded by 2 BC, etc. Thus, this calendar has no year 0, and the traditional time line is . . . , 2 BC, 1 BC, 1 AD, 2 AD, ...

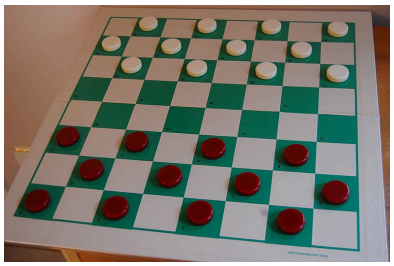

Figure 1: The starting position in Checkers [https://commons.wikimedia.org/wiki/File:CheckersStandard.jpg]

> 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.

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.

Write a white-box test of your classes.

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

Denne opgave går ud på at lave et såkaldt retro-style *roguelike* spil. Et roguelike går ud på at spilleren skal udforske en verden og løse nogle opgaver, ofte er denne verden et underjordisk fantasy *dungeon* befolket af monstre som skal nedkæmpes, og gåder der skal løses.

I denne opgave skal der arbejdes med at lave et objekt-orienteret design, som gør det nemt at udvide spillet med nye skabninger og spil-mekanismer.

Opgaven er delt i fire dele. I den første delopgave skal der arbejdes med at implementere en *canvas* i terminalen til at vise vores verden. Anden delopgave går ud på at lave et klasse-hierarki til at repræsentere skabninger og genstande i verden. Endelig skal der i den tredie delopgave arbejdes mod at sætte de forskellige dele sammen til et samlet spil. Fjerde del indeholde en række forslag til udvidelser, hvoraf I skal implementere mindst to.

I det følgende er der kun givet minimums-krav til hvilke metoder og properties I skal implementere på jeres klasser. I må gerne lave ekstra metoder eller hjælpe-funktioner, hvis I synes det kan hjælpe til at skrive et mere elegant og forståeligt program.

Rapport

Ud over jeres programkode skal I også aflevere en rapport (skrevet i LaTeX). I rapporten skal I beskrive implementeringen af jeres klasser, det vil sige hvilken skjult tilstand (interne variable og lignende), som jeres metoder arbejder på.

Ligeledes skal rapporten indeholde et UML diagram over klasserne i jeres løsning.

11g0 **Brugergrænseflade i Terminal**

For at vise spillets verden implementerer vi en klasse `Canvas`, som er et gitter af felter. Hvor feltet i øverste ventre hjørne har position $(0,0)$, og x-koordinatet tælles op mod højre, og y-koordinatet tælles op når man bevæger sig fra top mod bund.

Hvert felt har en `char`, og så *kan* feltet have en forgrundsfarve, og det *kan* have en baggrundsfarve.

Implementér klassen `Canvas` som har følgende signatur:

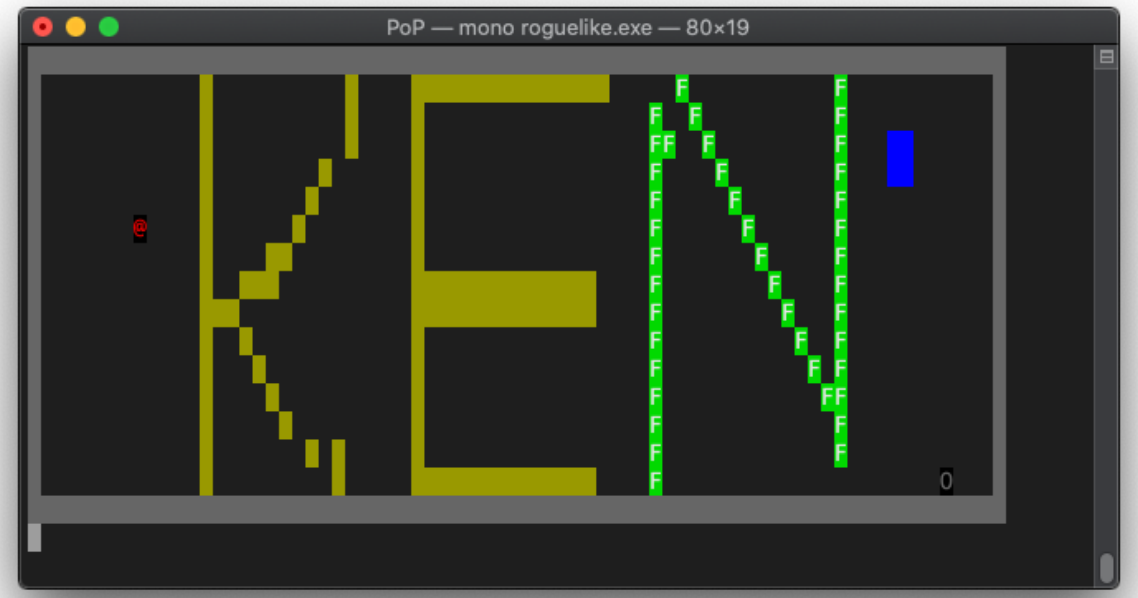

Figure 2: Eksempel på hvordan spillet kunne se ud i terminalen. Den røde `@` er spilleren, resten er genstande og skabninger som spilleren kan interagere med (måske med fatale følger).

```
type Color = System.ConsoleColor
type Canvas =
  class
    new : rows:int * cols:int -> Canvas
    member Set: x:int * y:int * c:char * fg:Color * bg:Color ->
   unit
    member Show: unit -> unit
  end
```

Det vil sige:

- En konstruktør der tager antal rækker og koloner som argumenter.
- en metode `Set` til at sætte indhold og farver på et felt.
- en metode `Show` til at vise en canvas i terminalen. Figur 2 viser et eksempel på hvordan det kunne se ud.

I rapporten skal I beskrive jeres designovervejelser, samt redegøre for hvilke data en canvas har.

Hints: Til `Show` skal I bruge følgende funktionalitet fra standard-biblioteket:

- `System.Console.ForegroundColor <- System.ConsoleColor.White` til at sætte forgrundsfarven til hvid (kan også bruges til andre farver).
- `System.Console.BackgroundColor <- System.ConsoleColor.Blue` til at sætte baggrundsfarven til blå (kan også bruges til andre farver).
- `System.Console.ResetColor()` til at sætte farverne i terminalen tilbage til normal.

11g1 **Genstande og Skabninger**

Vi bruger klassen `Entity` til at repræsentere genstande og skabninger i vores verden. Disse skal kunne renderes på en canvas. Tag udgangspunkt i følgende erklæring:

```
type Entity() =
  abstract member RenderOn : Canvas -> unit
  default this.RenderOn canvas = ()
```

Hvis I får behov for det må I gerne tilføje tilstand (data og properties), metoder og en anden default implementering af `RenderOn` til `Entity`.

Til at repræsentere spilleren bruger vi klassen `Player`:

```
type Player =
  class
    inherit Entity
    new : ...
    member Damage : dmg:int -> unit
    member Heal : h:int -> unit
    member MoveTo : x:int * y:int -> unit
    member HitPoints : int
    member IsDead : bool
  end
```

En spiller starter med ti hit points. En spiller er død hvis de har mindre end nul hit points. En spiller har et maksimum hit points de kan helbredes op til (I betemmer hvor mange, husk at dokumentere det i rapporten).

Metoderne `Damage` og `Heal` bruge til at gøre skade på og, henholdsvis, helbrede spilleren med et antal hit points.

Til at repræsentere genstande og skabninger, som spilleren kan interagere med, bruger vi den abstrakte klasse `Item`:

```
type Item =
  class
    inherit Entity
    abstract member InteractWith : Player -> unit
    member FullyOccupy : bool
  end
```

Den måde en spiller interagerer med et `Item` på, er ved at gå ind i `Item` (det kommer vi tilbage til i næste delopgave). Til dette skal vi bruge `FullyOccupy` til at sige om `Item` fylder feltet helt ud eller om spilleren kan stå i samme felt som genstanden. Metoden `InteractWith` bruges dels til at genstanden kan have effekter på spilleren, og dels så siger retur-værdien om genstanden stadigvæk skal være i verden (`true`) efter interaktionen, eller om den skal fjernes (`false`) fra verden.

Implementér følgende fem konkrete klasser der nedarver fra `Item`:

- `Wall` der fylder et helt felt, men ellers ikke har effekter på spilleren.
- `Water` der ikke fylder feltet helt ud, og helbreder med to hit points.
- `Fire` der ikke fylder feltet helt ud, og giver ét hit point i skade ved hver interaktion med spilleren. Når spilleren har interagereret fem gange med ilden går den ud.

- `FleshEatingPlant` der fylder feltet helt ud, og giver fem hit point i skade ved hver interaktion med spilleren.
- `Exit` vejen ud af dungeon!

11g2 Verden

Implementer klassen `World`:

```
type World =
  class
    new : ...
    member AddItem : item:Item -> unit
    member Play : unit -> unit
  end
```

Metoden `AddItem` bruges til at befolke verden med ting som spilleren kan interagere med. Typisk inden spillet går i gang.

Metoden `Play` bruges til at starte spillet, og tager sig af interaktionen med brugeren via terminalen. Spillet er tur-baseret og foregår på følgende vis:

(a) Vis hvordan verden ser ud, samt om der er eventuelt er sket noget for spilleren

(b) Hent brugerens træk som gives ved brug af pile-tasterne.

(c) Afgør hvilke `Items` som brugeren eventuelt interagerer med, samt hvad det betyder for hvad spillerens position og helbred er.

(d) Hvis spilleren er død eller hvis spilleren har fundet `Exit` vis et afslutningsskærmbillede og stop spillet, ellers start forfra.

Klassen `World` samt de andre klasser fra de andre delopgaver skal være i filen `roguelike.fs`. Lav derudover en fil `roguelike-game.fsx`, der som minimum laver en ny verden og kalder `Play`.

Basal Storyline

Den mest basale udgave af spillet: Spilleren starter et sted i et underjordisk dungeon, og skal finde udgangen. Når spilleren finder udgangen skal de have mindst fem *hit points* for at kunne tvinge døren op og undslippe dungeon.

Det er op til jer hvordan dungeon skal se ud, hvor spilleren starter, samt hvor mange genstande og skabninger der er i verden.

Hints:

- Det er en vigtig pointe at `World` ikke tager sig af at rendere spilleren og `Items` i verden, men blot skaber en canvas, som de forskellige `Entry` kan rendere sig selv på.
- Brug `System.Console.Clear()` at fjerne alt fra terminalen inden verden vises.
- Brug `Console.ReadKey(true)` til at hente et træk fra brugeren
- Hvis `key` er resultatet fra `Console.ReadKey` så er `key.Key` lig med `System.ConsoleKey.UpArrow`, hvis brugere trykkede på op-pilen.

Krav til afleveringen

Afleveringen skal bestå af:

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

Zip-filen `11g_<(gruppe)navn>.zip` skal indeholde en og kun en mappe `11g_<(gruppe)navn>`. I den mappe skal der ligge en `src` mappe og filen `README.txt`.

I `src` skal der ligge følgende og kun følgende filer:

- `roguelike.fs` og `roguelike-game.fsx`,

som beskrevet i opgaveteksten. Programmerne skal kunne oversættes med `fsharpc`, og de oversatte 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 ifølge:

- `Absalon->Files->noter->LaTeX->rapport.pdf`

guiden og oversat fra LaTeX. Husk at pdf-filen skal uploades ved siden af zip-filen på Absalon.

God fornøjelse.