# BDSA 2022 - Assignment 2

Laurits Kure, Frederik Petersen

September 22, 2022

Link to Github repo: https://github.com/thekure/group_17_assignment-02

# 1 C Sharp

## 1.1 Types

### 1.1.1 Class

- Defines responsibility
- Is a reference type.
  - Allocated on the heap.
  - Garbage collected.
  - Assigning big reference types is cheaper than that of value types.
  - An operation of one variable can affect another, since multiple variables can contain references to the same object.
- Supports inheritance.

### 1.1.2 Struct

- Is a value type.
  - Allocated on the stack.
  - Allocations are genereally cheaper than that of reference types.
  - Operations on one variable does not affect another, since each variable has its own copy of the data.
- Can have the same members as a class, excepting finalizers.
- Does not support inheritance, and thus cannot be marked as abstract or protected.
- Instantiation does not require use of 'new' operator.

### 1.1.3 Record Class

- Formerly known simply as records.
- Main purpose: storing data.
- Like classes, but immutable by default; it is designed to work well with immutable data.
- Reference-type.
- Supports nondestructive mutation; new records are created from existing ones, while classes modify existing states.
- Useful in creating types that combine or hold data.
- Overrides toString

### 1.1.4 Record Struct

- New in C10
- Instead of being like a class, these are like structs.
- Mutable by default; use readonly modifier to achieve immutability.
- Value-type.
- Unlike regular structs, these support == and != operators.
- Overrides toString
- According to benchmarks, about 20x faster than regular structs.

### 1.1.5 When to use which?

**If my data type**

- represents a single value, similar to a primitive type
- is immutable
- does not have to be boxed and unboxed frequently

...it should be a *struct*. Since record structs are 20 times faster, it should probably be a *record struct.*

**If not, but the data type**

- encapsulates a kind of complex value that is immutable
- is used in a one way flow

...it should be a *record class*. If none of the above applies, it should be a *regular class*.

## 1.2 Extension methods

We can implement the flatten methods from last week using the LINQ extension method `SelectMany()`. Using this we can flatten xs like so:

```
xs.SelectMany(i => i)
```

This returns a flattened enumerable of type T

The filter method can likewise be implemented using the LINQ extension method `Where()`, making use of the predicate parameter.
This means we can filter ys like so:

```
ys.Where(i => i % 7 == 0 && i > 42)
```

Which returns the filtered list

The same method can be used in conjunction with the leap year method implemented in assignment 0, to find leap years in a list of years:

```
ys.Where(year => isLeapYear(year)
```

Where, as we recall, `isLeapYear` is a method using this logic:

```
isLeapYear(year) => year > 1582 && (year % 400 == 0 || (year % 100 != 0 && year
                                    % 4 == 0))
```

This was implemented in assignment 0.

## 1.3 Delegates / Anonymous methods

The methods have been implemented in their corresponding tests in the Github repository.

The first method (reverse a string and print it) has been implemented by using the built-in ”Action” delegate
We instantiate the Action, with a lambda function, like so:

```
Action<string> rev = (string str) => {
    foreach (var c in str.Reverse())
    {
        Console.Write(c);
    }
};
```

Product method is implemented in much the same way. First we define, however, as it should have a return value, we instead use the built-in ”Func” delegate
We instantiate it, with a lambda function, defining the multiply behaviour:

```
Func<int, int, int> p = (int one, int other) => one * other;
```

Lastly we repeat the process for the last method (the comparison of a string containing a numeric value, and an integer).
We use the Func delegate this time as well
We then instantiate it with a lambda function:

```
Func<str, int, bool> compr = (string str, int val) => Int32.Parse(str) == val;
```

It should be noted, that this doesn't handle strings, that don't contain a numeric value, as this wasn't a requirement of the assignment

# 2 Software engineering

## 2.1 Exercise 1

### 2.1.1 Use Case

Use cases are descriptions of interactions between users and a system, often using a graphical model and a structured text. A use case identifies the actors involved in an interaction. A great use of use cases is during the design-process of a software system. The use cases identify core functionality, responsibilities and necessary user access.

### 2.1.2 Scenario

A scenario is basically a user story, which in turn is basically a series of use cases. From Sommerville:

> *Stories are written as narrative text and present a high-level description of system use; scenarios are usually structured with specific information collected such as inputs and outputs.*

Scenarios provide the baseline for use cases. They are often used in the formulation of requirements.

## 2.2 Exercise 2

### 2.2.1 User Requirements

Natural language statements about the system-services-expectations of the system-users, and about the constraints in which they must operate. Typically informal and easily understandable to end-users, and can vary from broad statements to precise descriptions.

### 2.2.2 System Requirements

These are specified with more detail than the user requirements, and contain detailed descriptions of the system expectations, such as functions, services and constraints. From Sommerville:

> *The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented.*

### 2.2.3 Functional Requirements

These describe how a system should function, and sometimes how it should not function. From Sommerville:

> *These are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations.*

### 2.2.4 Non-Functional Requirements

These are (often system-wide) constraints on functions and system-services, including but not limited to:

- Time
- Finances
- Standards
- Formats
- Constraints development process

## 2.3 Exercise 3

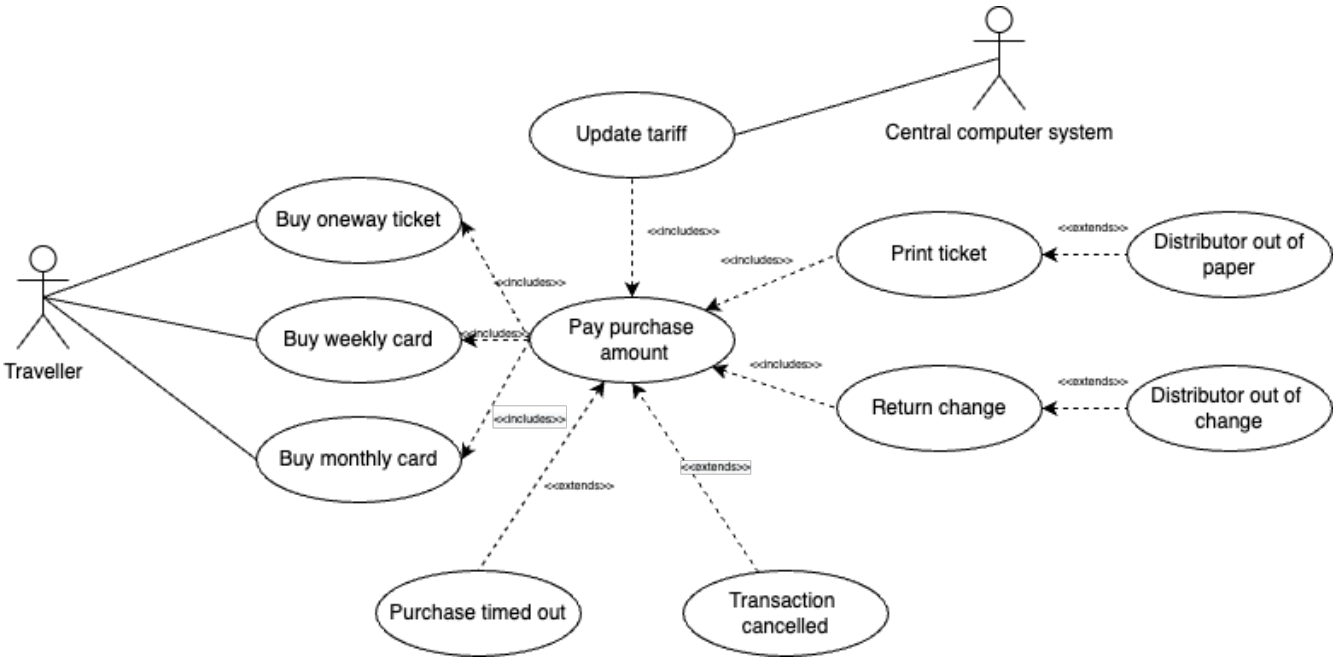


Figure 1: Use-case diagram made in draw.io

## 2.4 Exercise 4

### 2.4.1 Ambiguous Formulations

- Store dele af løsningen kan afvikles på mobile enheder; this does not specify exactly what functionality needs to be available to mobile users, and what can remain desktop functions. If it is all of them, the requirement should say so.
- En stabil og driftsikker løsning..; we can make assumptions about what exactly is meant by this, but definitions of stability and reliability can be subjective depending on the context.
- ...skal den digitale understøttelse være medvirkende til, at kommunen er et attraktivt sted at arbejde; in what way does the solution exude attractiveness to the employee? Attractiveness is a relative and subjective term.

### 2.4.2 Missing Information

Considering the requirement is simply entitled ”Brugervenlighed”, it could be argued that such descriptions have no place here, but reading the requirement, it is impossible to extract what kind of program the client is actually looking for. It only states in vague terms, that it needs to help the employees in their day to day tasks. What are these tasks? Who are these employees? What does user-friendliness mean to the client? In what sense is the solution meant to ”guide” the employees?

### 2.4.3 Problematic Formulation of Non-Functional Requirements

- Store dele af løsningen kan afvikles på mobile enheder.
- Medarbejdere skal digitalt understøttes i udførelsen af deres kerneydelser.
- En stabil og driftssikker løsning er en medvirkende faktor til større tilfredshed med ansættelsen i kommunen.
- Derfor er det vigtigt, at store dele af løsningen kan afvikles på mobile enheder.

### 2.4.4 Rewrite

The requirement attempts to cover a lot of ground, so we have divided it into smaller parts with different headlines. Since amending a lot of the ambiguity and missing information requires information which is impossible for anyone but the client to provide, a lot of the rewritten requirements is left blank.

1. Brugervenlighed
   (a) *// This is where the requirement should describe exactly what the client means by user // friendliness.*
   Løsningen skal guide vores medarbejdere i at løse deres opgaver, ved at *** INDSÆT FORKLARING HER.
2. Målgrupper
   (a) *// This should be a finite list of the clients current target audience, instead // of a vague statement.*
   Kommunen har en række forskellige faglige målgrupper, som alle skal tilgodeses af den fælles løsning. Disse målgrupper er *** INDSÆT ALLE MÅLGRUPPER HER ***. Nogle af disse målgrupper kræver mobil adgang til løsningen. Derfor skal løsningen have en mobil udgave, som inkluderer ***INDSÆT SPECIFIK RELEVANT FUNKTIONALITET HER.
3. Attraktivitet ift. fastholdelse af medarbejdere
   (a) Undersøgelser for attraktiviteten for ansættelse i det offentlige til dels forstærkes når arbejdspladsen har digitale løsninger, som *** INDSÆT RELEVANT DATA HER. Dette bør være en del af løsningen.
4. Stabilitet og Driftsikkerhed
   (a) En stabil og driftssikker løsning er en medvirkende faktor til større tilfredshed med ansættelsen i kommunen. For os betyder stabilitet og driftsikkerhed, at ***INDSÆT SPECIFIK FORKLARING HER. Dette skal inkluderes i løsningen.

## 2.5 Exercise 5

### 2.5.1 Actors

An obvious actor is a musician wanting to use the the tracker to produce music. Assuming there are no pure software systems implemented, this would be the only way to compose full arrangements, without

having to play every instrument themselves.

A sound system or speaker could be considered an actor, as it needs to access the music from the music tracker, potentially even accessing it in the form of a MIDI.

### 2.5.2 Use cases

- The music tracker needs to support multiple tracks in a pattern, for different instruments. Optionally a single track should support multiple instruments.
- The music tracker should be able to apply an effect to a step, changing the sound accordingly
- The music tracker should be able to alter the amount of steps and tempo of a pattern, shortening or lengthening the pattern accordingly.

### 2.5.3 Non-functional requirements

1. The system must adhere to the 12-tone scale, and be in tune with what is considered to be correct for that tonality.
2. The system should have high response time, to allow adding sounds to steps, as they are being played.
3. The system should utilize no more memory than what can be provided by the hardware of the tracker.

## 2.6 Exercise 6

```
UC1.
Pay for a Bagel
Main Actor:  Customer
Main Scenario:

  1. Customer initializes screen
  2. System shows multiple options for product categories
  3. Customer chooses the lunch option
  4. System shows lunch products
  5. Customer chooses Bagel
  6. System adds the Bagel product to basket
  7. Customer chooses Pay with credit card
  8. System enables terminal with the price of one bagel
  9. Customer scans credit card
 10. System returns to start-screen
```

**Requirements**

1. Functional: System must support payment system.
2. Functional: System must have a basket system, that adds up prices for the payment.
3. Functional: System must go back to start-screen after successful payment.