BDSA Assignment 2

Starcke, Viggo
vist@itu.dk

Roager, Mads Aqqalu
mroa@itu.dk

September 22, 2022

1 C#

Link to Github repo: Github-link

1.1 Types

All four different constructs in C# are all some sort of data types. Here we differ *value types* from *reference types*. In short value types are copies of their assigned value and stored on the stack while reference types refer to their assigned value and are stored on the heap (their reference is stored on the stack).

Class: A *class* is a reference type. Classes have all the functionality of inheritance, polymorphism, plus the fields can be initialized and so on. So in that way you haven't restricted yourself in any ways when using classes. On the other hand it doesn't have any nice built-in features, and sometimes there are smarter ways to go about constructing objects than classes, if one knows the purpose of these. For comparison classes only compare the reference object. One has to implement their own method if a value-based comparison is desired. Same goes for ToString, and so on.

Struct: A *struct* is a value type. Structs have a bit of a narrow scope of usage compared to the class construct. Many have their different rules of when to use a struct instead of a class, they all take different motives and it can be hard to navigate. But a few things to consider before choosing a struct construct is you data types main responsibility. Typically a struct is suited for data storage with a small memory footprint. And also take into consideration if your data type needs inheritance (have subclasses) or if it will ever be polymorphic. In the end a lot of memory usage can be saved by using structs over classes, but only if done carefully.

Record Class: A *record class* differs from the class construct in quite a few ways. First of all it is immutable, meaning it cannot change state after it has been created. It is still considered a reference type by default, just like a class. But the important part is that it has a lot of generated methods. A built-in value-based equality check for example checks if the types of two records are equal and then proceeds to check all the values. A record also has a ToString(), printing a nice representation of the record type, and a nice way to deconstruct the record. Following is a simple example of deconstructing a record.

```
1        public record Student(int Id, string GivenName, string Surname);
2
3        var queenElizabethII = new Student(2, "Queen", "Elizabeth");
4        var (id, givenName, surname) = queenElizabethII;
```

Altogether you get less boilerplate code, which in large projects might save you from a lot of repetitive code.
A good using scenario for a record class is as a way of conveniently grouping related bits of data together.

Record Struct: A *record struct* is much similar to the record class with the big exception that it is mutable. This can be changed with the readonly keyword.
The record struct has the value type construction and good usage for data storage from struct, while also getting the built-in methods from a record. In short; a struct with a ToString() and value-based equality checker who can also operate under terms like '!=' and '=='.

1.2 Extension Methods

1. `xs.SelectMany(i => i);`
2. `ys.Where(i => i % 7 == 0 && i > 42);`
3. `ys.Where(i => DateTime.IsLeapYear(i))`

1.3 Delegates

1.3.1 String reverse

```
1 public delegate void PrintReverse(string input);
2 PrintReverse printReverse = delegate(string input) {
3           var charArray = input.ToCharArray();
4           Array.Reverse(charArray);
5           var reversedString = new string(charArray);
6       };
```

1.3.2 Multiplying Decimals

```
1 public delegate decimal Multiply(decimal d1, decimal d2);
2 Multiply multiply = delegate(decimal d1, decimal d2) {
3             return d1*d2;
4         };
```

1.3.3 Whole number and string

```
1 public delegate bool WholeNumberAndString(int i, string input);
2 WholeNumberAndString wholeNumberAndString = delegate(int number, string
      input) {
3             input.Trim();
4             int parsedString = int.Parse(input);
5             return number == parsedString;
6         };
```

2 Software Engineering

2.1 Exercise 1

Describe the difference between a scenario and a use case. Describe for each of the two concepts when and for what they are used.
A **scenario**, is, in a way, a story, describing a specific situation step-by-step where the use of the proposed software system is practical. A scenario describes in detail, what the user knows, does, and finds out during the scenario. Scenarios are often used to show a stakeholder what a system is good for, and for specifying requirements, as a scenario is easy to relate to. E.g what would be the right solution to certain scenario etc.
Use cases, can be described as a set of scenarios, that is illustrated through a graphical model and/or structured text. They give an overview of the system and it's actors, and are especially used in system design, rather than requirements engineering.

2.2 Exercise 2

Identify and briefly describe four types of requirements that may be defined for a computer-based system.
In general, the difference between requirements are not always easily described, as they can often overlap into different categories.

2.2.1 User Requirements

User requirements describe, what a system should provide for the end-user, and under which constraints and circumstances. They describe the functionality of the system on a high level.

2.2.2 System Requirements

System requirements describe more in depth what the software system should provide. User requirements may form several system requirements, detailing how a certain user requirements should be fulfilled.

2.2.3 Functional Requirements

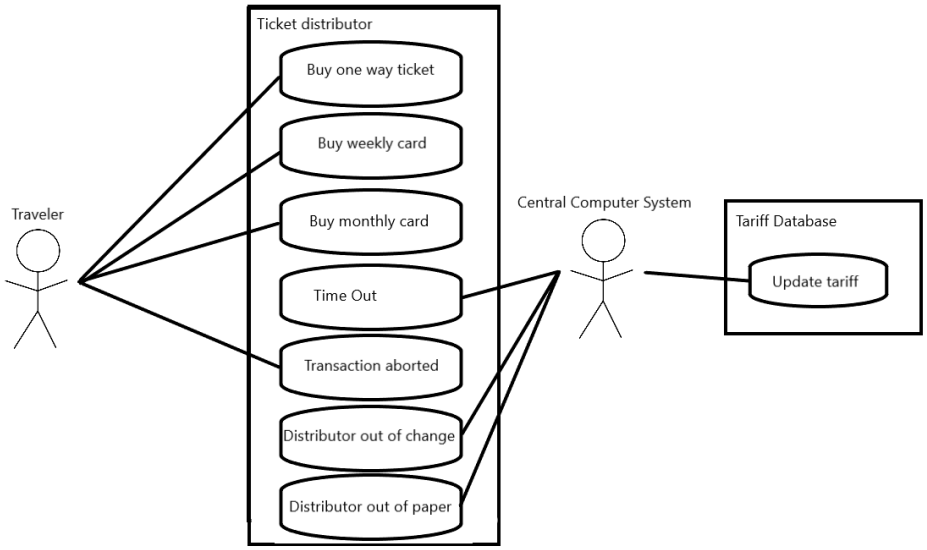
Functional requirements describe, what the software system should do. Sometimes very specifically, for example, what output a specific input should return, or perhaps what it should not return. Sometimes they are more general, about the functionality of the system.

2.2.4 Non-functional Requirements

Non-functional requirements describe the constraints that the software is built under. This could be constraints regarding the memory usage or time complexity, or the system could be required to follow some standard for data usage, or something different. In general, non-functional requirements only deals with stuff around the solution, and not the functionality of the system.

2.3 Exercise 3

Draw a use case diagram for a ticket distributor for a train system. The system includes two actors: a traveler, who purchases different types of tickets, and a central computer system, which maintains a reference database for the tariff. Use cases should include: Buy OneWay Ticket, Buy Weekly Card, Buy Monthly Card, Update Tariff. Also include the following exceptional cases: Time-Out (i.e., traveler took too long to insert the right amount), Transaction Aborted (i.e., traveler selected a cancel button without completing the transaction), Distributor Out of Change, and Distributor Out of Paper.



2.4 Exercise 4

Discover formulations in the requirement that are ambiguous.
Here are examples of ambiguous formulations:

- "Medarbejdere skal digitalt understøttes i udførelsen af deres kerneydelser, og derved skal den digitale understøttelse være medvirkende til, at kommunen er et attraktivt sted at arbejde."
- "Det er altafgørende, at medarbejderne føler sig godt understøttet af den nye digitale løsning."
- "hvilket tilsammen kan understøtte en faglig stolthed."
- "at løsningen er medvirkende til god lovmedholdelighed i sagsbehandlingen"

Is there any information missing in the requirement?
As stated at the top of the requirement, it is supposed to be about a good user experience, and ease of use. This is hard to read out of the requirements contents themselves.
The employees core output is used in a formulation, but what the core ouput is, is missing.
The requirement describes that good lawfullness is important, but without being a municipal employee, it is hard to know what this lawfullness is about. *The requirement specifies a set of non-functional requirements. What is problematic about there formulation?*

The requirements are mentioned, but need to be described in much further detail. This leads to them being ambiguous and hard to understand. *Rewrite the requirement according to what you identified as problematic in the three bullet points above.*
The requirement should probably be rewritten into several other requirements, where all the parts could be explained and described in further detail. The requirement should be much shorter. A proposition for this requirement is: "Medarbejdere skal bruge mange funktioner og megen information til at udføre deres arbejde, der foregår i mange forskellige omstændigheder. Derfor er det vigtigt, at løsningen er brugervenlig, og har en overskuelig og let tilgængelig brugergrænseflade. Dette er med til at lette den enkelte medarbejders daglige byrde, og gøre kommunen et mere attraktivt sted at arbejde."

2.5 Exercise 5

Identify actors that interact with a music tracker software system:

Generally speaking are actors external to the software system itself and will therefore in the case of a music tracker software be something like:

- User
- Some sort of library or database (of instruments and effects)
- Storage database
- Speakers or similar, handling the audio output

Formulate three use cases in structured language that a software music tracker system has to support:

UC1: A user wants to start a project running on a length of 16 bars.

- User chooses 'start new project'.
- Music tracker system creates up new empty project file and prompts user for presets.
- User chooses project criteria of length 16.
- Music tracker system puts presets into the newly created empty project file.

UC2: A user wants to change the instrument of a cell 2c from kick to bass.

- User navigates to cell 2c.
- User chooses the 'instrument' option.
- Music tracker system prompts a pop-up list of available instruments.

- User chooses bass.

UC3: A user wants to fill cell 2a-2d with a random euclidean pattern from D3 to G6 on a chromatic scale.

- User marks cell 2a-2d
- User chooses til 'fill' option
- Music tracker system prompts a pop-up window where settings for the 'fill' action can be managed
- User selects 'Euclidean' and 'Random' under 'Pattern'-section, 'Chromatic' under 'Scale'-section, 'D3' under 'From'-section and 'G6' under 'To'-section.

Express three non-functional requirements for a music tracker software system: Non-functional requirements refer to a software systems non-functional properties or qualities. In this case they could look like:

- The music tracker software should be adaptable, and usable with all kinds of computers. System requirements should be low while maintaining high functionality.
- The music tracker software should be fast and easy in using it. Latency is non tolerable in giving our costumers the best experience possible.
- The music tracker software should have a backup security system, so that no project is ever lost if power supply cut, a crash or other unforeseen problems occur.

2.6 Exercise 6

Use case that ITU canteen has to support: A costumer wants to get a portion and handle the checkout themselves paying with card.

- Costumer picks a plate and fills it with desired food.
- Costumer goes to a checkout stand, queuing may be necessary.
- Costumer places plate on weight and selects the fitting category.
- Canteen system outputs a price and prompts a few paying options.
- Costumer selects 'VISA/Debit card'.
- Canteen system readies the credit card machine.
- Costumer pays the amount via card, either by contactless or chip.
- Canteen system returns a receipt.

System requirements for the above use case: For the above use case to be fulfilled to a satisfactory level following system requirements has to be met. (Many others occur, this is just three examples)

(Functional requirement) The canteen system has to take weight-input from the external weight and convert it to property which it then can pass to both the GUI and the credit card machine.

(Functional requirement) The canteen system has to format data from the purchase compact and nicely into a small script which it both outputs as a receipt and stores internally.

(Non-functional requirement) The canteen system has to work nice and smoothly hindering long queues during rush on lunch breaks. This includes communicating fast and easy with the external weight.