1 GitHub repository link

You should be able to click a link right here, otherwise the link will be displayed just below, for you to copy:

`https://github.com/andreaswachs/BDSA2021-AS01`

2 C#

2.1 Generics

The type constraints for the first version of `GreaterCount`, it constraints the type `T` to be implementing the `IComparable` interface, ensuring that items of type `T` can be compared between themselves. The generic type `U` is not having any constraints applied to it, but it is neither used in this case.

Additionally, the `item` object type needs to be implementing `IEnumerable` with the generic type `T` provided.

The type constraints for the second version of `GreaterCount`, we achieve the same result as the above answer, but through transitivity where the constraints are that `T` is of same type of `U` and type `U` implements the `IComparable` type with type `U`. This is only accepted if the `IComparable` interface allows for implementations for the type `U`.

3 Software Engineering

3.1 Exercise 1

> *"Knowledge acquisition is not sequential"* - Bruegge & Dutoit

Bruegge and Dutoit state that the process of acquiring knowledge is not a linear process. We are not going to learn everything that we will from a software project in a nice, linear manner. They also state that a single piece of information can reset the estimated remaining effort that a software project needs in order to be done.

An example of this stems from my own 1st year software project. This was a project about drawing the map of Denmark from XML data from Open Street Maps. We had implemented data structures to save spacial data and were working on path finding, to find the shortest path between two point on the map. We got done implementing the shortest path algorithm

based on Dijkstra's shortest path algorithm with the A* optimization. When immediately implemented we thought that we had finished all the work in regards to finding paths on the map, but we discovered a large amount of edge cases that came to life from working with the complete map of Denmark. The effort into making the path finding algorithm behave *correctly* stretched on for quite some time after we expected to be done with all work in regards to path finding.

3.2 Exercise 2

> *"The ticket distributor is composed of a user interface subsystem, a subsystem for computing tariff, and a network subsystem managing communication with the central computer."*

This decision were made during the **requirements elicitation**. This decision makes an abstract claim of how the ticket distributor is composed, but doesn't describe technical details.

> *"The ticket distributor will use PowerPC processor chips."*

This decision were made during the system design phase. The decision dives into a very specific technical detail.

> *"The ticket distributor provides the traveler with an on-line help."*

This decision was made during the requirements elicitation. This decision is still very abstract and doesn't go into technical details.

3.3 Exercise 3

The term *account* is used as an application domain concept for only the first sentence:

> *"Assume you are developing an online system for managing bank accounts for mobile customers"*

Here it is explained how it fits together in the solution (that it is managed by the system that the software engineers will develop).

Right after the first sentence, the text uses the concept of an *account* in the solution domain as it is now being discussed how to solve some practical problems that affect the solution domain (lack of network connectivity):

> *"A major design issue is how to provide access to the accounts when the customer cannot establish an online connection. One proposal is that accounts are made available on the mobile computer, even if the server is not up. In this case, the accounts show the amounts from the last connected session"*

3.4 Exercise 4

On the difference between building airplanes, bridges and software.

When building airplanes and bridges, you firstly build them such that they obey physical law that needs to be kept in order for them to operate. Bridges needs to be able to carry the weight of all the traffic that crosses it; the airplane needs to be sufficiently aerodynamic and powerful, such that it can gain flight. While building airplanes is a modern endeavor, building means of transportation has been around for ages. Building bridges has been done since the dawn of civilization.

When building software, there are not in the same way these physical bounds. These physical products doesn't have to conform to physical laws of aerodynamics. They have no physical shape. A bridge or an airplane will quickly be deemed fit to work when built, because its flaws are visibly apparent - this is not the case for software. For example, we can have deadlocks and "off-by-one" errors in the hiding for a long time before they are discovered where they might invalidate near uncountably many person hours.

Building software is also difficult because they are mostly built to serve a purpose in a certain application domain that a customer exists in. The people who make the software are not necessarily part of that application domain - and they have to model a solution domain to model the application domain. The validity of the software products is therefore limited by the software developers understanding of the application domain.

3.5 Exercise 5

1. *"The TicketDistributor must enable a traveler to buy weekly passes."*

   This is a functional requirements.

2. *"The TicketDistributor must be written in Java."*

   This is a non-functional requirement.

3. *"The TicketDistributor must be easy to use."*

   This is a non-functional requirement.

4. *"The TicketDistributor must always be available."*

   This is a non-functional requirement.

5. *"The TicketDistributor must provide a phone number to call when it fails."*

   This is a functional requirement.

3.6 Exercise 6

"What is the purpose of modeling?"

The purpose of modeling is to represent a complex system, in part or in whole, at the appropriate level of abstraction. By this, we can hide a lot of technical details that might get in the way of an overview that we'd like to have of a system.

This is useful for software engineers that needs to develop software systems to be used in application domains that are outside of their primary field of work. This means that we can model application domains with just enough knowledge to be able to craft a software system for it.

The modeling of the application domain leads to the opportunity to be able to model the solution domain, with the given object oriented methods given in the Object-Oriented Software Engineering (Bruegge & Dutoit, 2010).