Cpt S 321 – Final Exam

Spring 2022

30 points total

Total pages: 4

Read the instructions carefully until the end before asking questions:

- This is an individual exam: you are **not allowed to communicate** with anyone regarding the exam questions.
- If something is unclear, ask for clarifications via the Canvas Discussion (no code allowed). If it is still unclear after my answer, then write down any assumptions that you are making.
- At the end, make sure you download your exam code in a clean directory to ensure that it works.
- No late submissions are allowed.
- **What to submit (failure to follow the submission instructions below will result in receiving 0 automatically for the exam):**

1. In your **in-class exercises** GitLab repository create a branch called "FinalExam" and commit i) a .PDF version of your design document where you also indicate the good design practices you use and clarify any additional design choice you deem necessary and ii) your code. Tag what you want to be graded with "FinalExam_DONE" tag. The design document must be placed in a folder "Design_Documents" at the root of your project and the name must contain your name and a description of the diagram (ex.: "VeneraArnaoudova-ClassDiagram.pdf").
2. Your GitLab readme file must:
    - explain what features you implemented for the exam and the ones that you are missing.
    - provide a link to a short video capturing your screen where 1) you show us how you download your code from the GitLab repository in a clean directory, and 2) you execute your application from that clean download and you show us that it runs.
3. On Canvas, submit a link to your GitLab final exam.

Please read the entire description carefully before you start working.

You are contacted by a small educational company to build a desktop application in C# with a GUI (using WinForms) that will help teach young children different properties of food items starting with fruits and vegetables. The idea is the following: Children can create food items (fruits or veggies) and assign them different properties (color, shape, texture, size, taste). Children can also create containers (food baskets that are initially empty) that they can fill with existing food items of one type only (i.e., a basket cannot contain a mix fruits and veggies but mixing different fruits for example is allowed).

Here is a summary of the features that you need to design and implement for the prototype:

1. **Create a food item**: Children should be able to create a food item (either a fruit or a veggie) and select values for color (ex. yellow, red, green), shape (ex. square, circle, oval), texture (ex. firm, soft, crunchy, chewy), size (ex. tiny, small, medium, large, huge), taste (ex. sour, sweet, bitter).
2. **Change an existing food item**: Children should be able to select an existing food item and change its properties. Multiple properties can be changed at once.
3. **Create a container (or a food basket)**: Children should be able to create a container and indicate what type and how many food items this container can hold. The max limit should not exceed 10 items for this prototype to facilitate showing the food items and baskets in the application.
4. **Add a food item to a container**: Children should be able to select an existing food item and an existing container and add the food item to the container. If the food item and container are not compatible or if the container is full a proper message should be shown.
5. **Remove a food item from a container**: Children should be able to select an existing food item from a container and click on a button to remove the food item from the container. The food item is then free to be reassigned to another container.
6. **Delete a food item**: Children should be able to select an existing food item either from a container or an unassigned food item and delete the food item after which the food item no longer exists.
7. **Delete a container**: Deleting a container means that the container and any food item that it contains will no longer exist.
8. **Undo last action**: Children should be able to undo the last performed action (which can only be one of the above listed actions).

The application contains multiple tabs (or views) as follows:

1. **A first tab for creating and setting up the food items and baskets** that contains all the necessary controls to allows children to perform all the features listed above. In addition, this form shows all containers and their content and all food items that have not yet been placed in a container. This means listing all items waiting to be placed in containers (with all their properties) and all containers, the types of food they contain, their limits, and the food items they contain (with all their properties).
2. **A second tab in which children can apply filters on existing containers**. This form shows all containers and their content and allows children to apply filters on the different container properties such as “what are the fruit baskets” or “what are the baskets with more that 3 food

items". The result is shown in addition to showing all containers and their content as we want children to always see what the current state is and what the outcome of the filter will be.

3. **A third tab in which children can apply filters on existing food items**. This is similar to the previous form with filters except that here the filters are applied on the food items. For example, "what are all yellow food items" or "what are all food items that are sweet". Again here the result is shown in addition to showing all containers and their content.

Note that all views show the current state of existing baskets and food items and thus all views must be updated when that state changes and you should use best practices to do so. For displaying the state of the application, you do not have to use graphics – i.e., you can use text elements displaying a fruit that is created with the appropriate characteristics. For implementing the filtering you are asked to use LINQ.

Grading schema and point breakdown (30 points total):

- **5 points**: A design document showing a class diagram that represents your design. Your diagram must be annotated with all Design Patterns and Principles that you use. Feel free to add other types of diagrams if need be. Feel free to use any software (such as draw.io) to make the class diagram – don't forget to export it as a .PDF to include in your submission.
- **10 points**: Your software fulfills all the requirements above with no inaccuracies in the output and no crashes.
- **3 points**: For a "healthy" version control history, i.e., 1) the prototype should be built iteratively, 2) every commit should be a cohesive functionality, and 3) the commit message should concisely describe what is being committed.
- **4 points**: Code is clean, efficient and well organized.
- **2 points**: Quality of identifiers.
- **2 points**: Existence and quality of comments.
- **2 points**: Existence and quality of test cases. Normal cases and edge cases are both important to test.
- **1 point**: summary of the features implemented and features missing (if any).
- **1 point**: Video showing that you run the application from a clean download.

General Homework Requirements	
Quality of Version Control	• Should be built iteratively (i.e., one feature at a time, not in one huge commit). • Each commit should have cohesive functionality. • Commit messages should concisely describe what is being committed. • Use of a .gitignore. • Commenting is done alongside with the code (i.e, there is commenting added in each commit, not done all at once at the end).
Quality of Code	• Each file should only contain one public class.

	● Correct use of access modifiers. ● Classes are cohesive. ● Namespaces make sense. ● Code is easy to follow. ● StyleCop is installed and configured correctly for all projects in the solution and all warnings are resolved. If any warnings are suppressed, a good reason must be provided. ● Use of appropriate design patterns and software principles seen in class.
Quality of Identifiers	● No underscores in names of classes, attributes, and properties. ● No numbers in names of classes or tests. ● Identifiers should be descriptive. ● Project names should make sense. ● Class names and method names use PascalCasing. ● Method arguments and local variables use camelCasing. ● No Linguistic Antipatterns or Lexicon Bad Smells.
Existence and Quality of Comments	● Every method, attribute, type, and test case has a comment block with a minimum of <summary>, <returns>, <param>, and <exception> filled in as applicable. ● All comment blocks use the format that is generated when typing "///" on the line above each entity. ● There is useful inline commenting in addition to comment blocks that explains how the algorithm is implemented.
Existence and Quality of Tests	● Normal, boundary, and overflow/error cases should be tested for each feature. ● Test cases should be modularized (i.e, you should have a separate test case for each feature or scenario that you test - do not combine them into one large test case).