D2 explanation

For the initial subsystem decomposition for the deliverable 2 code it seemed logical to organize the subsystems based on their functionality. Regardless of whether we are working with Client or Animal profiles, we would always be using a similar approach for creating, viewing and adding them to storage. For this reason, creating subsystems dedicated to Adding and Viewing these profiles seemed to make the most sense.

Furthermore, having a single subsystem dedicated entirely to the local and persistent storage of these profiles appeared was the obvious solution considering they all functioned in relatively the same way.

D2 limitations

While this approach seemed to work initially, when mapping the interactions between subsystems it became apparent that there were some significant drawbacks in decomposing the system in this manner.

Firstly, even though the classes in each subsystem performed the same task, the classes had absolutely no relation to each other. Therefore, we had AddAnimal and AddClient in the CreationSubsystem even though the adding an Animal had absolutely no relation to adding a Client and vice versa. This resulted in our subsystems having very low cohesion with the majority of subsystems not having any functional relations between classes.

Secondly, because we had such low cohesion the functional relations between our classes ended up happening between subsystems. This led to situations where the execution of even the simplest task would make use of many subsystems. This meant that as a result of the low cohesion within each subsystem, our system also had exceptionally high coupling.

[FIGURE 3004\_subsystem\_decomp\_class.xml]

Goals for improvement for entire system

Considering the deliverable 2 subsystem decomposition had some major problems with regards to cohesion and coupling, our main goals for the subsystem decomposition for the final product was to create subsystems that perform similar tasks in such a way where the objects are functionally related. This way the system could operate with minimal between-subsystem interaction.

Explanation/Evolution of design choices

When overlooking the deliverable 2 subsystem decomposition we noticed that a significant cause of low cohesion within our subsystems was that Animal-specific classes and Client-specific classes never interacted. Utilizing this observation we were able to conclude a better approach would involve organizing the subsystems based on the type of Profile with which they are interacting.

For this reason, we decided to create an AnimalSystem that included Animal, AddAnimal, ViewAnimal and AnimalStorage. Similarly, we also created a ClientSubsystem that included Client, AddClient, ViewClient and ClientStorage. Both Add and View objects made significant use of their respective Storage objects and by taking this approach we are able to keep all these relations contained to only one subsystem. This severely limited the amount of relations that occurred between subsystems which resulted in much looser coupling and much greater cohesion within these specific subsystems.

We also created a PersistentStorageSubsystem that is separate from the AnimalStorage and ClientStorage objects. The majority of our system makes use of the AnimalStorage and ClientStorage objects while the DatabaseStorage object is rarely used. Therefore, creating a separate StorageSubsystem that contained exclusively these three Storage objects resulted in the majority of our system having to interact with the StorageSubsystem at some point in order to accomplish even the simplest of tasks. By separating PersistentStorage from AnimalStorage and ClientStorage, interaction between other systems and the Storage-specific subsystem only ever occurs when a Profile is created, edited or at program startup to load all of the database information.

Finally, with regards to the MainMenuSubsystem, we expanded the objects into a variety of subcategories based on the subsections with which they interact. This significantly increases code maintainability where we can make a modification to any given subsystem and have it only affect a single section of the MainMenuSubsystem. This allows us to reduce coupling between subsystems because we only have a small portion of the user menu directly related to each subsystem as opposed to the approach used in deliverable 2 where the entirety of the MainMenuSubsystem was functionally related to all of the subsystems.