Applied Software Engineering

Assignment2

Stefan M Ahmed

21359035

01/21

Intro

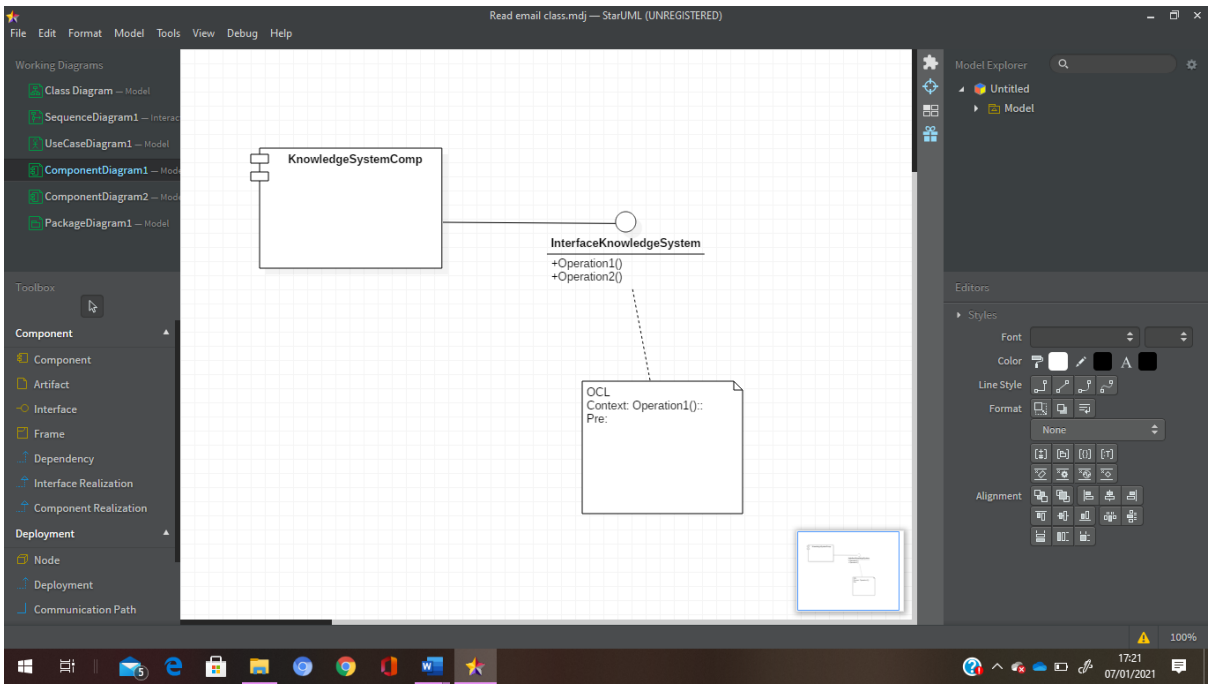
Hill & Knowlton the public relations firm focus their work on projects. This is done with their knowledge management system. They had problems with their knowledge management system and so had to find ways to fix this problem and get a new system employees would work with.

After finding reasons why the system was not worked with that well the firm was able to find a new system to manage and share information. This was as an extranet hk.net where employees could go to emails to read emails of projects done in the past to complete the projects they were currently doing. There were emails that had to be put in and stored in order as well as searched and put in the index.

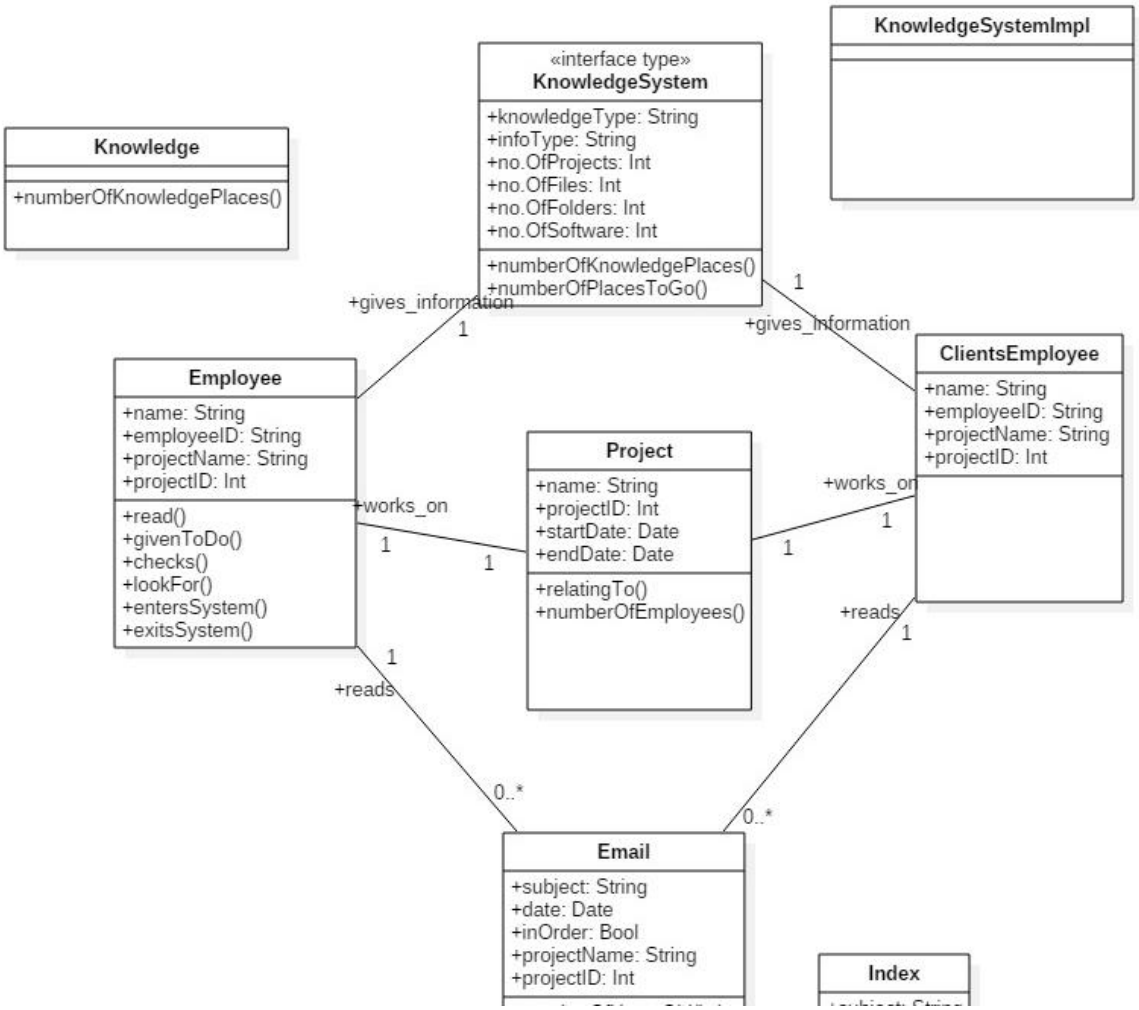
This coursework is about composing components. This is putting components together as a continuation of the class diagram of the read emails use case. Components represent the classes for the use case – read emails of past projects put in the archive.

Task 3

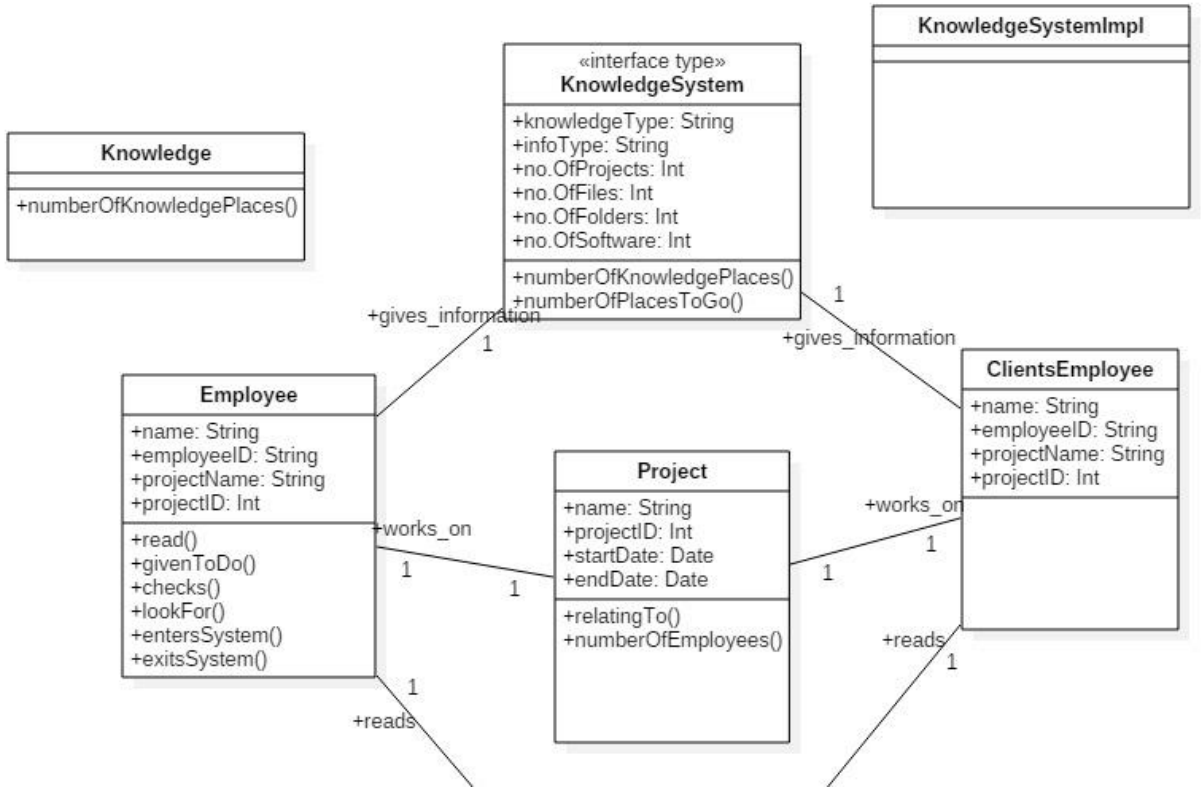
For this task the composition starts with the KnowledgeSystem class as a component. This component having the interface with the methods.



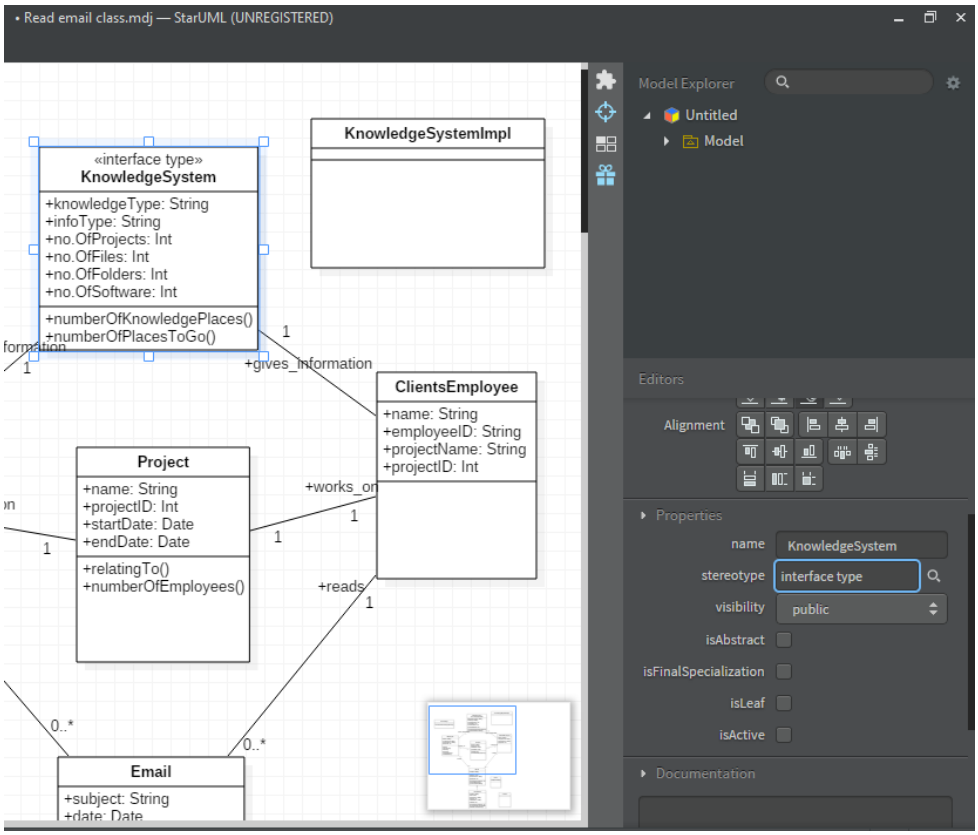
Operations are added to this interface. These being the associated interface methods.



In the class diagram the knowledgeSystem class I have stereotyped as <<interface type>>. It is stereotyped as that instead of just <<interface>>.

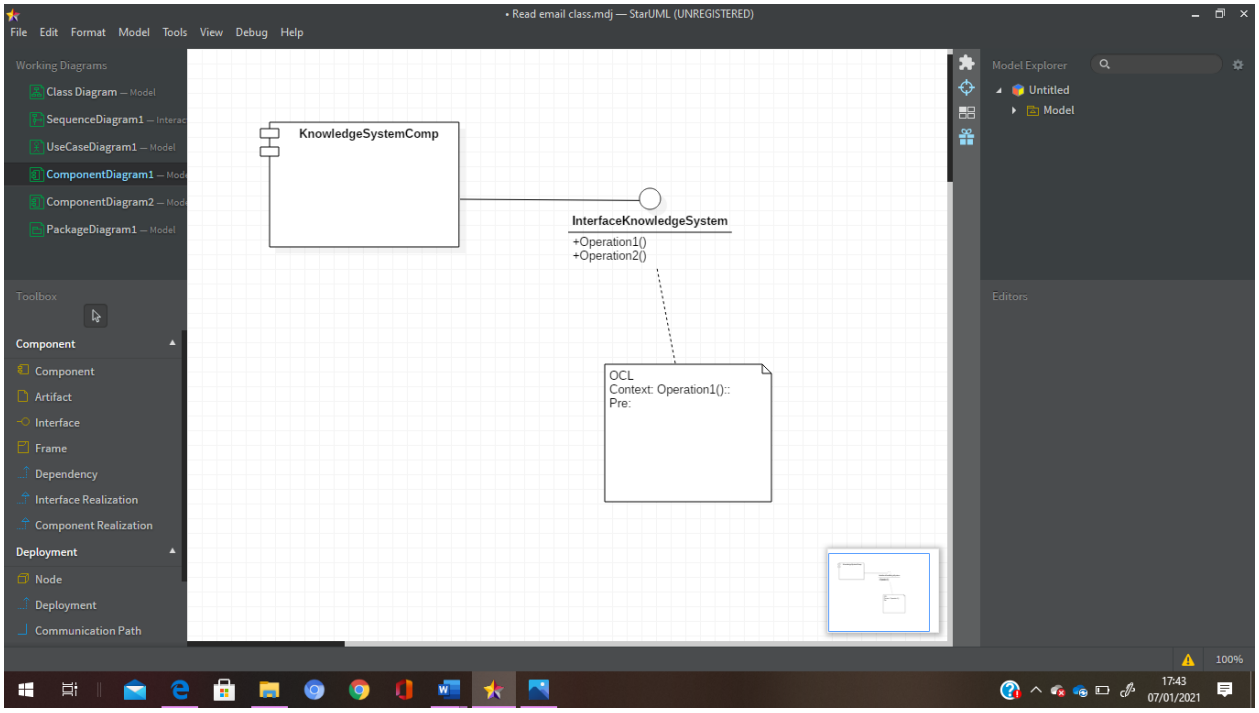


I then add the KnowledgeSystem implementation class with “Impl” followed by its name. This class represents the component of the KnowledgeSystem, being the component of KnowledgeSystem.



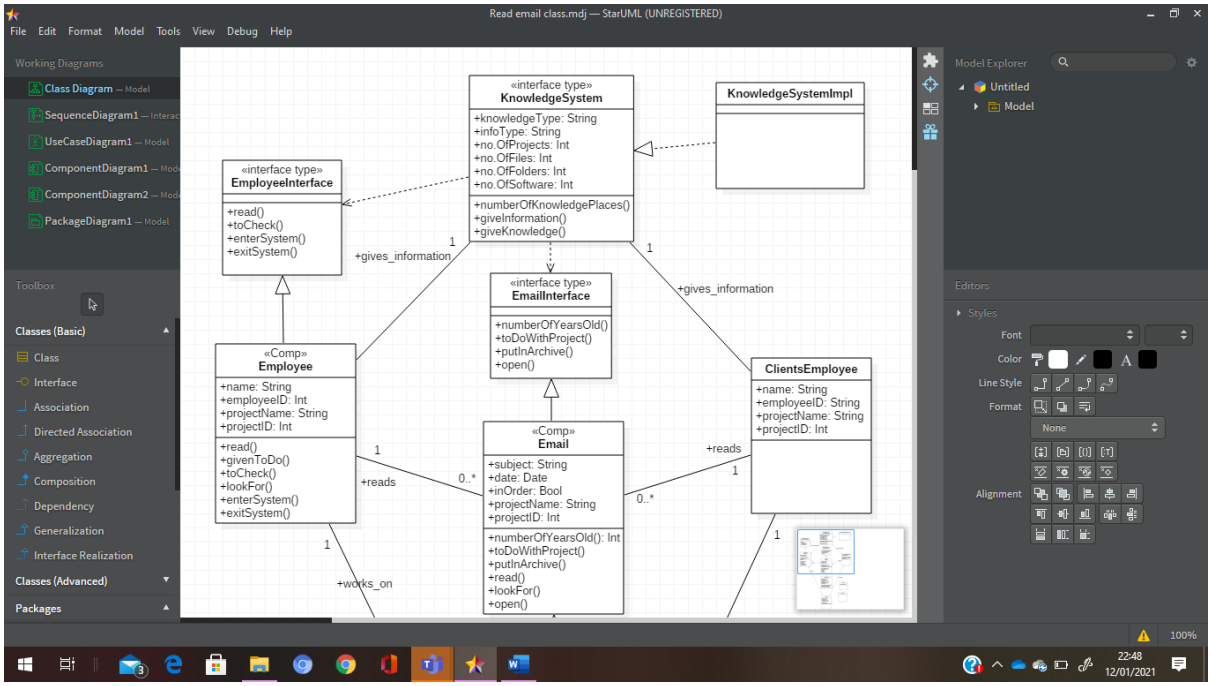
The stereotype is put by clicking on the class and putting “interface type” in the stereotype text box in the classes properties section. This lets us know it is the interface component.

Component Diagram

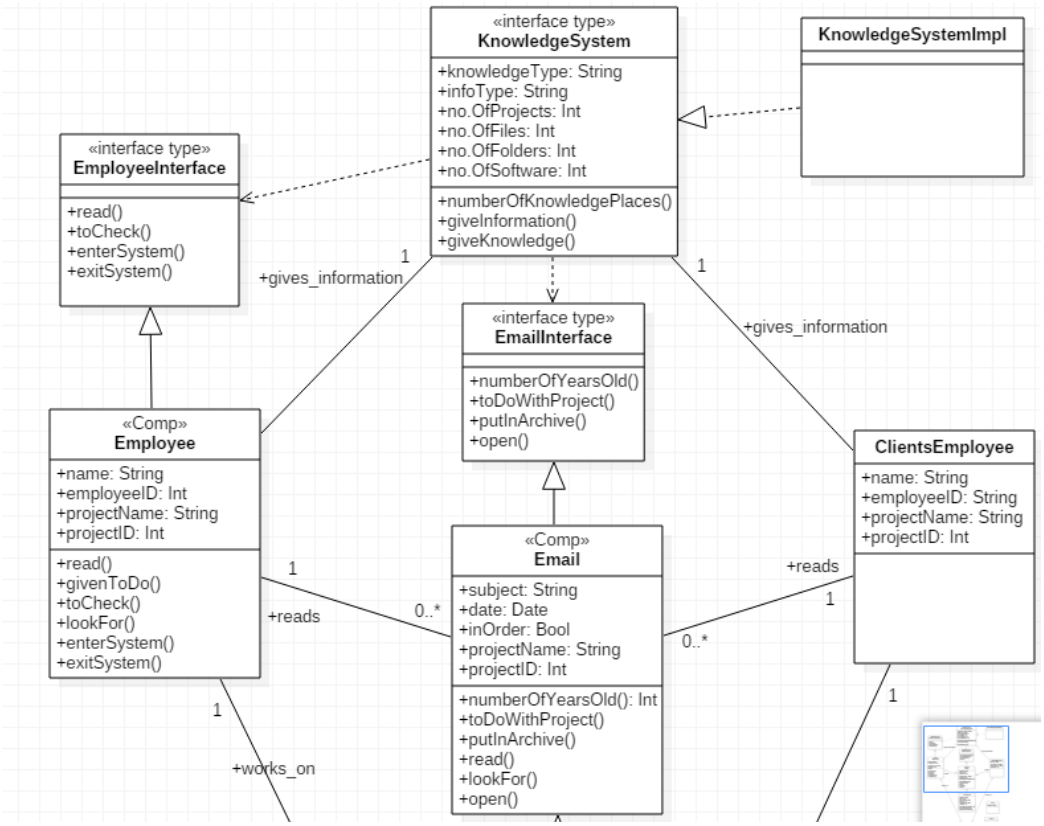


The component being known as “Comp”. The interface beginning with “Interface” followed by “KnowledgeSystem” rather than “KnowledgeSystemInterface”.

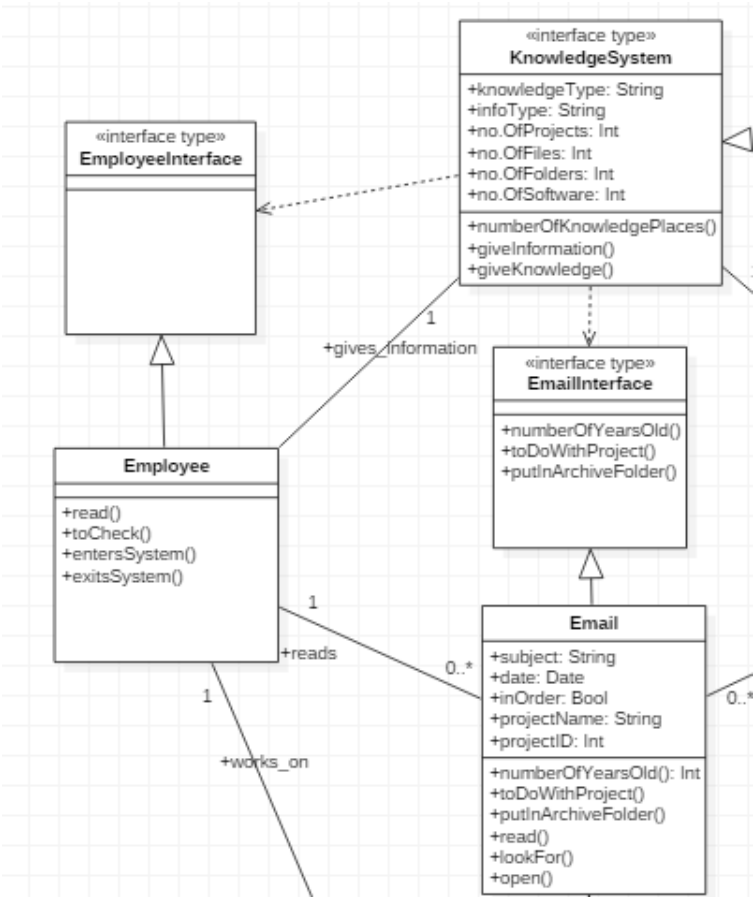
The interface has operations which are the methods. The methods belonging to the KnowLedgeSystem class go there if necessary. A link note to the interface, with its operations, gives the OCL. The conditions for the first operation Operation1() stating what the situation must be before the next situation happens.



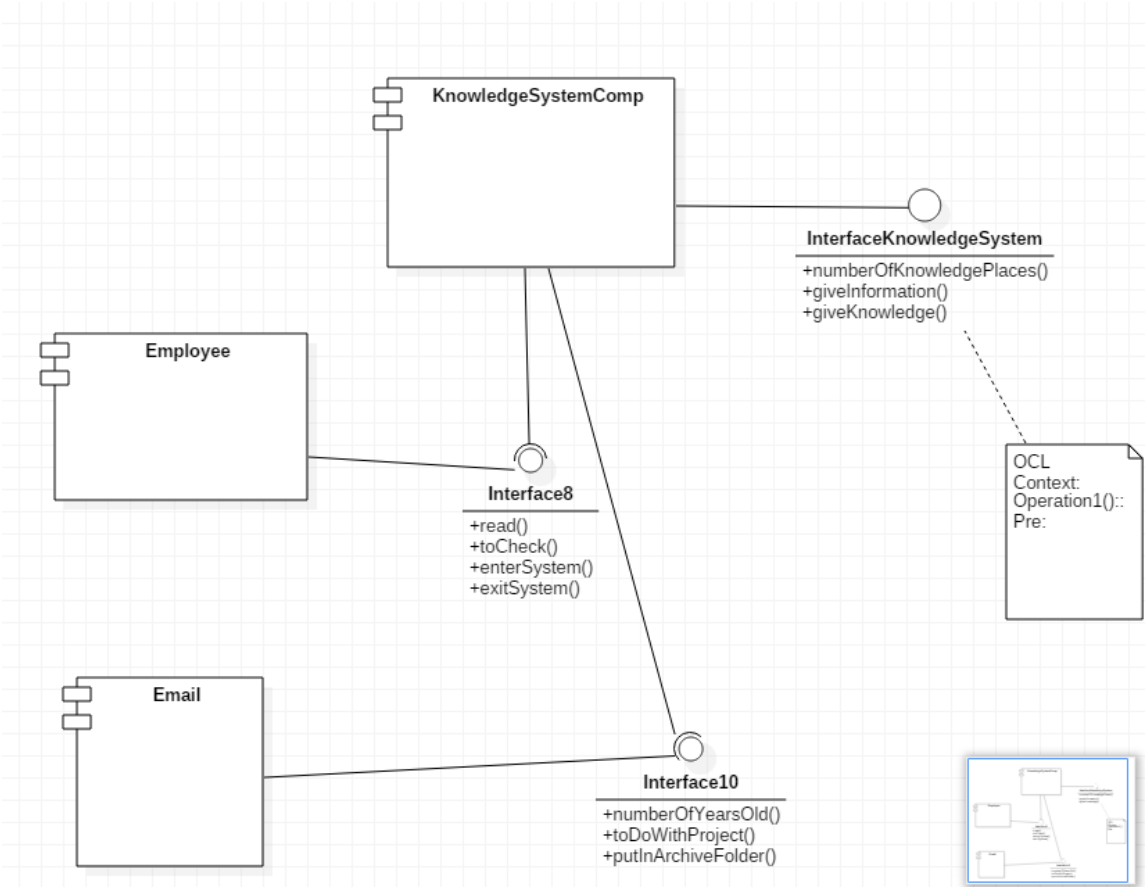
I add the rest of the stereotypes to the classes. These are Employee and Email each with their interface class. Employee and Email are stereotyped <<Comp>> to let us know they are the components. They are put together to communicate with each other in the component diagram. The idea of this is being to make a system that tells us the ways employees work with the system that provides and shares information. Information provided as emails the employee looks for and reads to help themselves with the project. This continues the work I did on the class diagram.



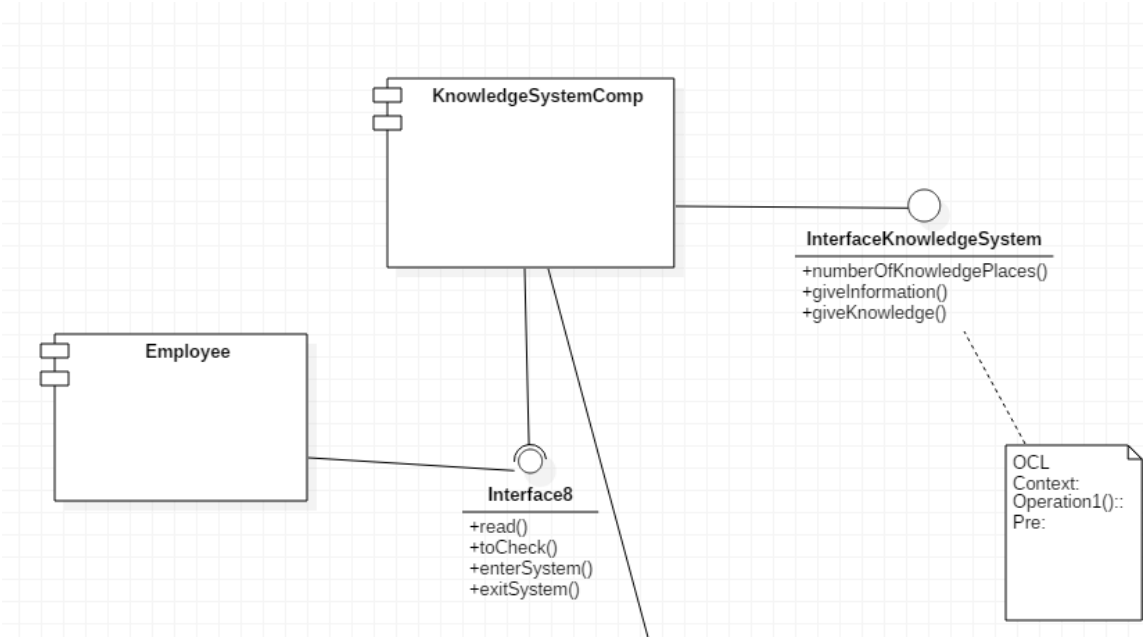
So here's the class diagram being updated to form the component model. This is by making copies of the classes but giving them stereotypes of the component they are in in the component diagram. I have also named the classes that are the interfaces of their components. I have connected each component class to their interface class with a generalisation to point out



Employee is to be a component with its interface. This component will provide a service through its interface which include enter the knowledge system and read the emails. The service given by the methods. These are the operations read() and enterSystem().

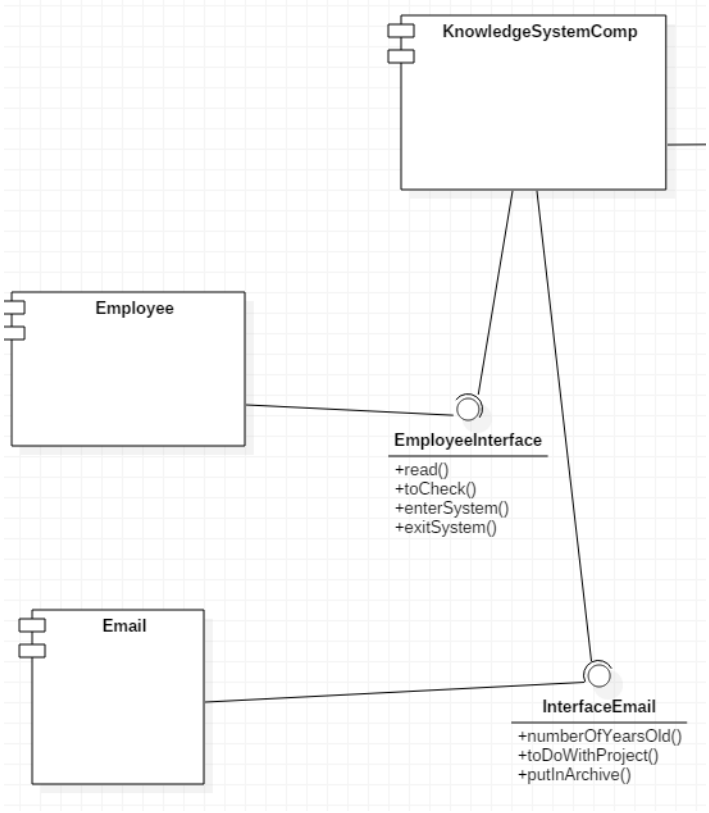


I now assemble the components together. This is the hierarchical composition – one component communicating with the other to provide a service. I put the Employee component with its interface being the employee entering the system to read emails. Having been given the project to work, on they read emails of projects done before, to help them do it. They also check details to make sure the project the email relates to helps them with the one they are currently doing.

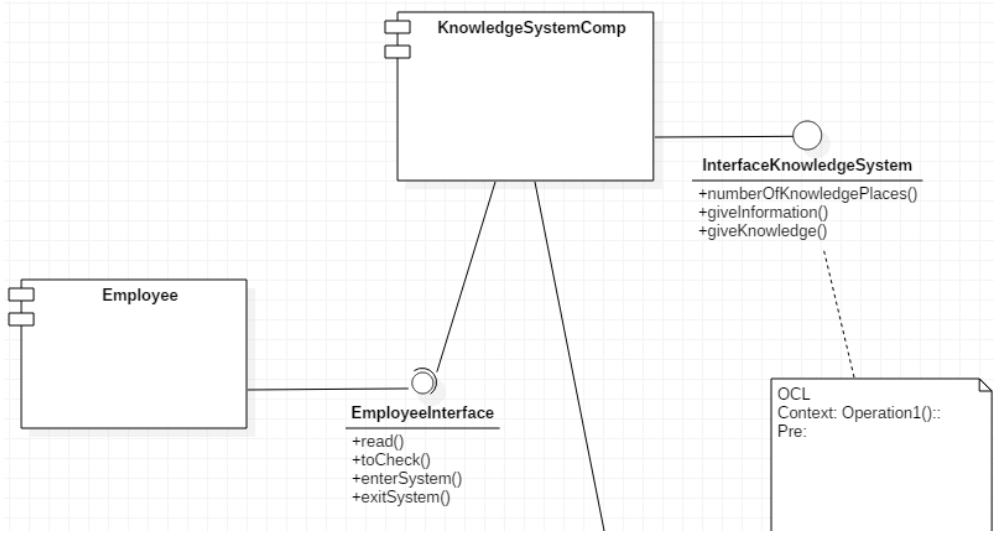


Employee is giving the service through its interface. KnowledgeSystemComp is executed next. Its interface comes handy with its numberOfKnowledgeP () method. Employees expect to get to

knowledge in one place to share it rather than in more. The giveInformation() and giveKnowledge operations() give the necessary information and knowledge to be shared, on the screen. Employees, on the screen, share information and knowledge with their team working on the project and other employees creating slightly different products.



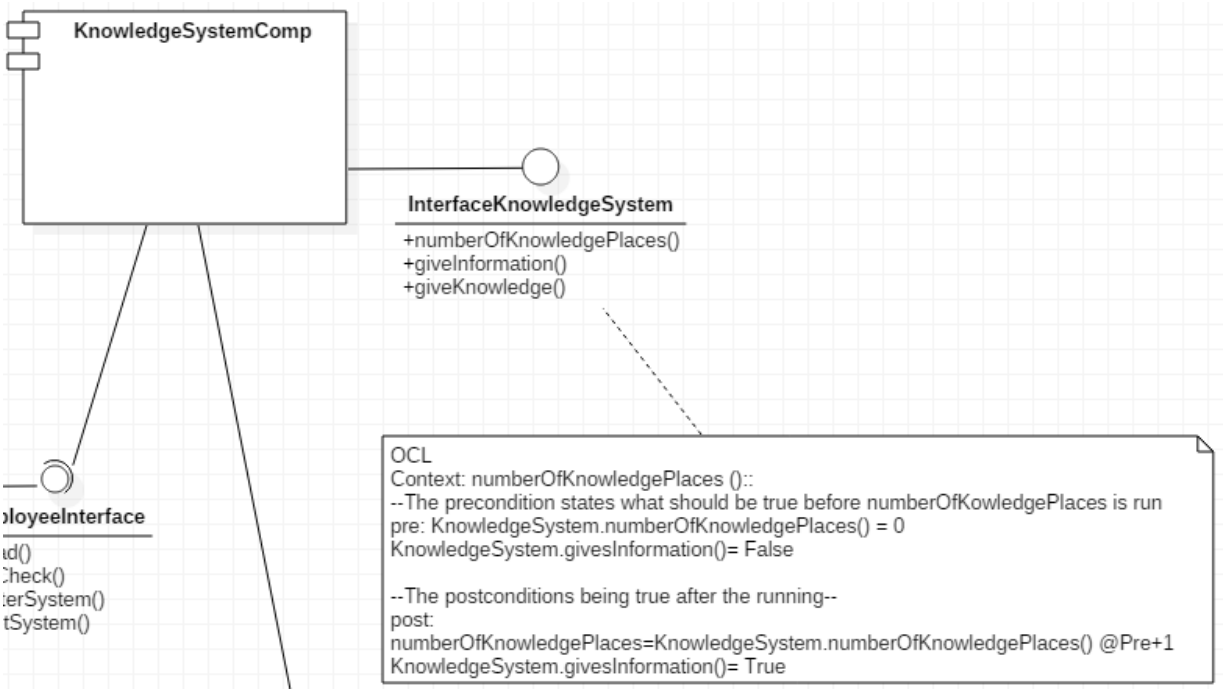
Having entered the system to get to the information given by it which is the emails of projects done before. The email component is executed next. The email component gives through its interface the services that include how long ago the email was sent, which project it is to do with, if they are put in the archive(as Boolean).



So in turn the employee is looking at the graphical user interface to press a button on the keyboard to go into the knowledge system and get information. Doing this This calls the

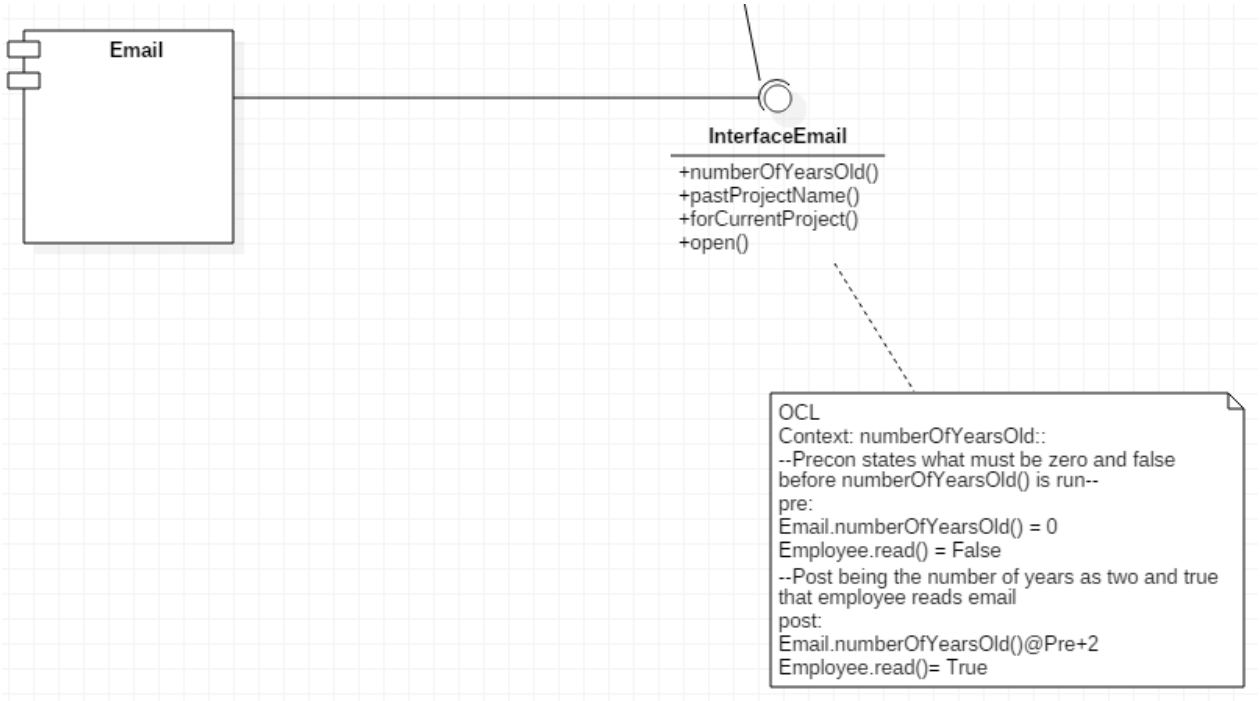
KnowledgeSystem Comp. The comp in turn asks the methods in the Employee interface to do their tasks to give the service i.e. invokes them.

The same with the Email. The employee in turn sees the GUI in front of them and presses a button to go into the system to get and read information as emails. Doing this This calls the KnowledgeSystem which in turn requests the methods in Email interface to carry out their tasks that give the service. This being the use case.



The OCL for the no.OfKnowledgePlaces in the KnowledgeSystem says:

- The should be no places yet to give the knowledge
- It is not true that the system gives information yet
- After the execution the number of places to provide the knowledge goes up by 1
- It then becomes true that the KnowledgeSystem gives information

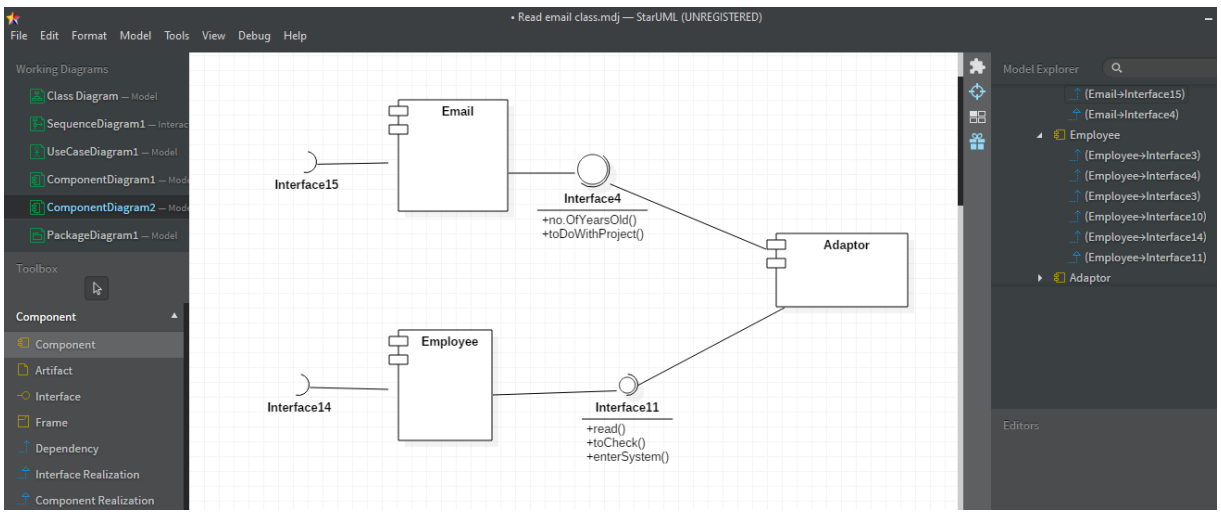


The OCL for the number of years ago the email was sent says:

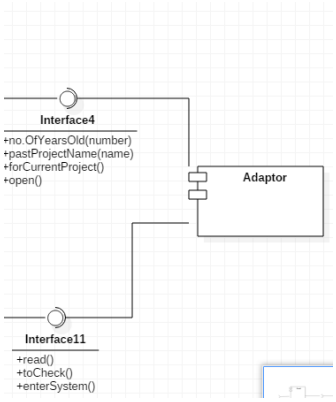
- The number of years should first be zero(could be greater than or equal to zero but being zero is straightforward here)
- This means the employee has not read it yet
- The number of years being added by two at the precondition means the employee reads the email

The other kinds of ways these components are composed are sequential and additive.

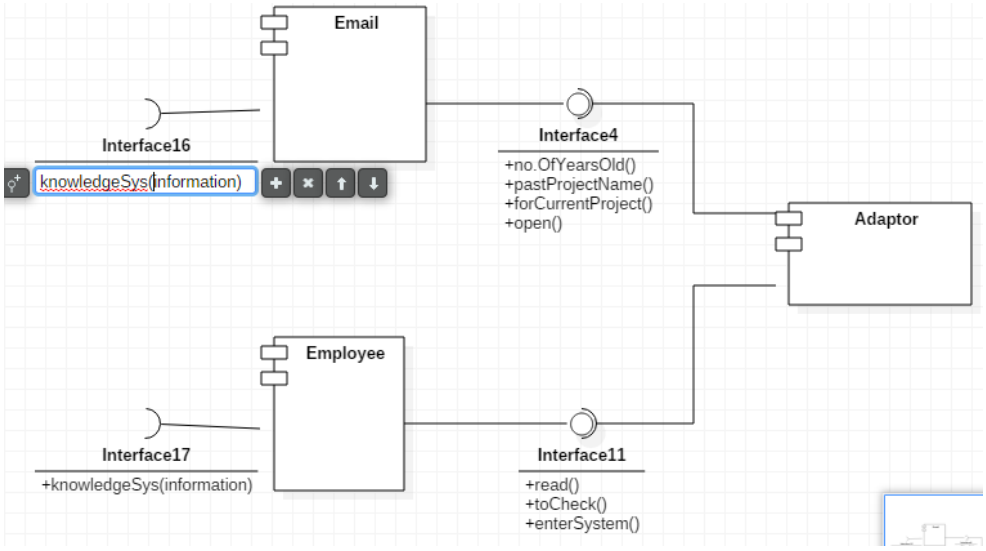
The sequential type of composition is with an adaptor. Data is passed through the adaptor from one component to the other. The components are put together to communicate with each ot…



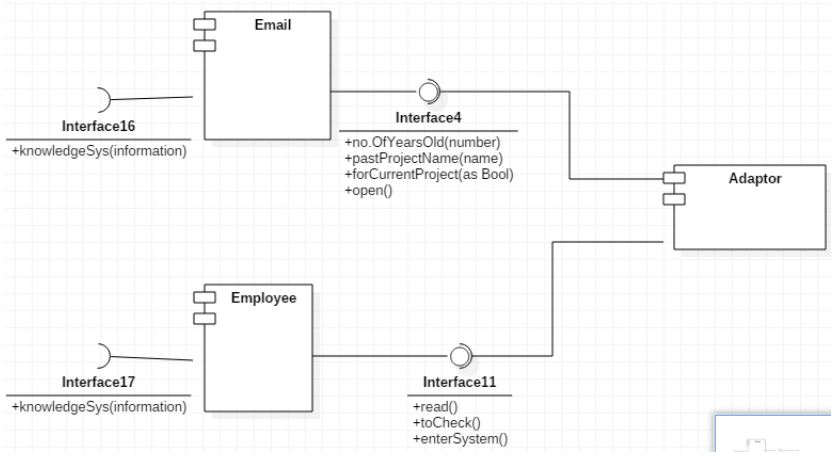
I first drag the two components which are Email and Employee. Double click on them to give them an interface with methods. The methods that are needed for this system are in the interfaces of the components the service is provided through. I then drag another component to be the adaptor. I name it Adaptor. If we put a component as a system, there are a number of classes to form the component. I have these two communicate with each other for now.



I add a dependency to the Email and Employee interface. So the two components are put together by the adaptor. Data passes through the adaptor from one component to the next. I have edited the methods too.

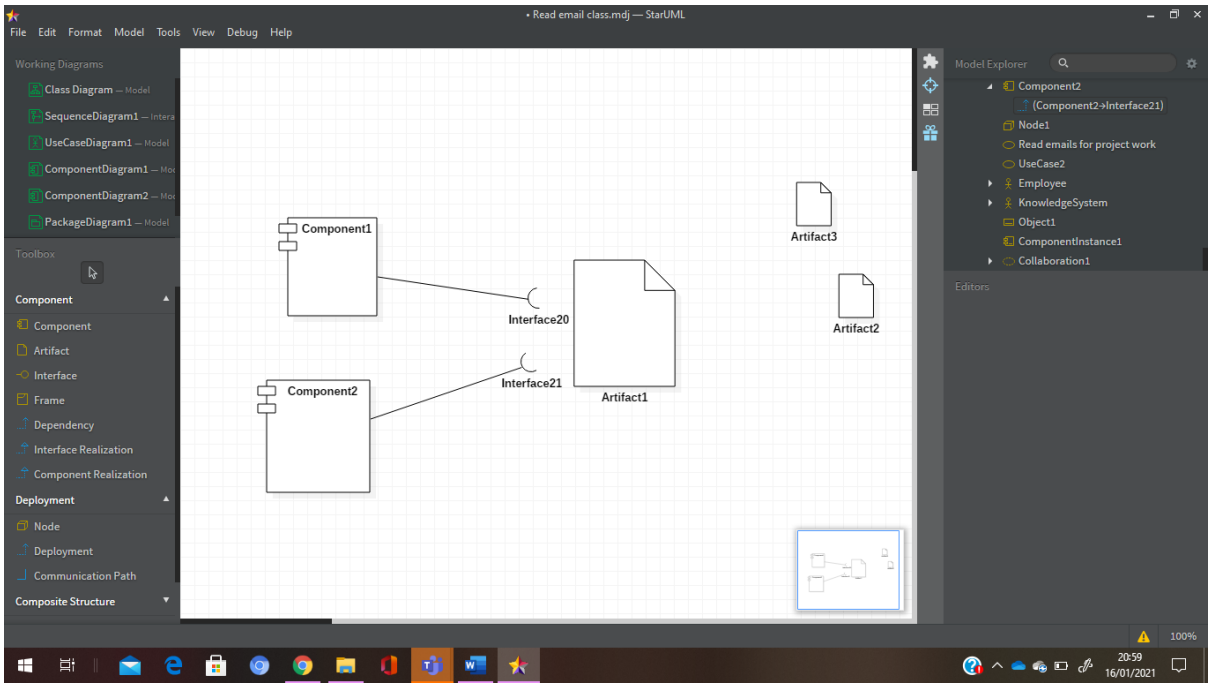


Now I add dependancies to connect Email and Employee to the Knowledge System. Connecting them to that is worth it as the Knowledge System has the information to give to them. I put “information” as its parameter which it is worth having. This being the number of years old, past project name, for current project which the employee works on the project with.



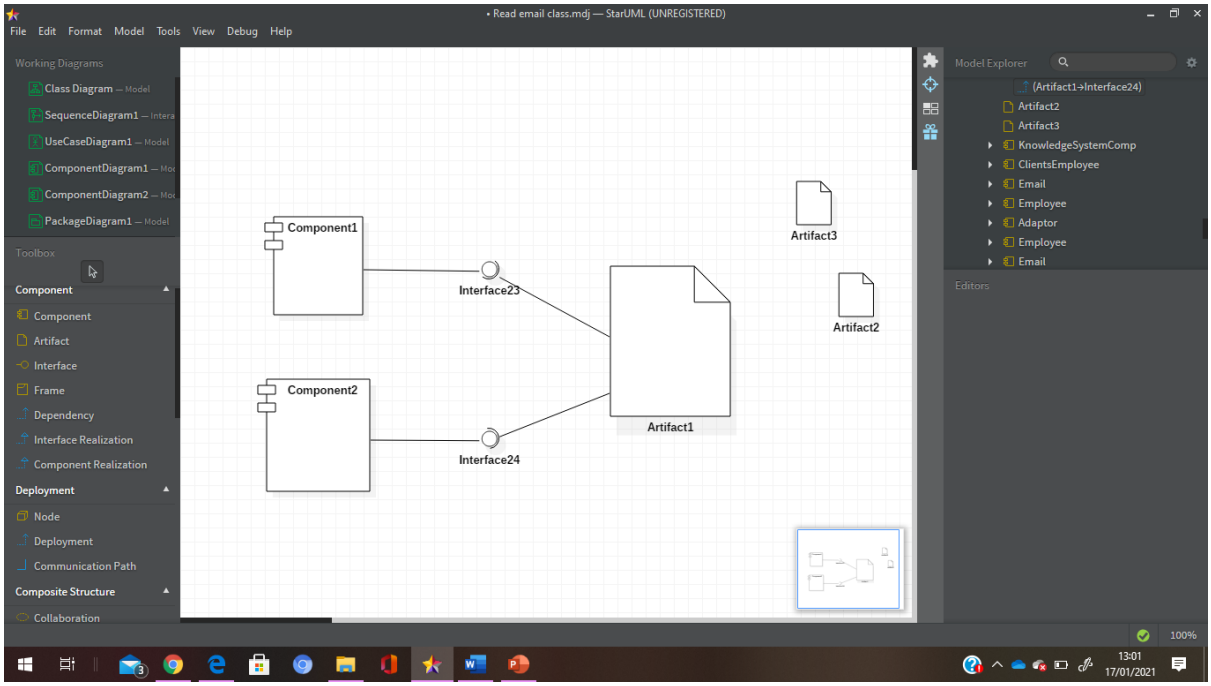
So this is the sequential communication. The Email and Employee are put together by the adaptor. The adaptor takes the Boolean value of the forCurrentProject() method. This method which is whether or not the email is for the project the employee is currently working on. It takes that and passes it to the Employee component. The service given is the information to help the employee with the project they have been given to work on. Information about past projects to help them with the one they are doing now.

The other kind of possible composition is the additive. Basically there a several components being together here. The types of interfaces that are given the purpose of are here. These are those that let other components have services and those that give the purpose of the services that they need to have.

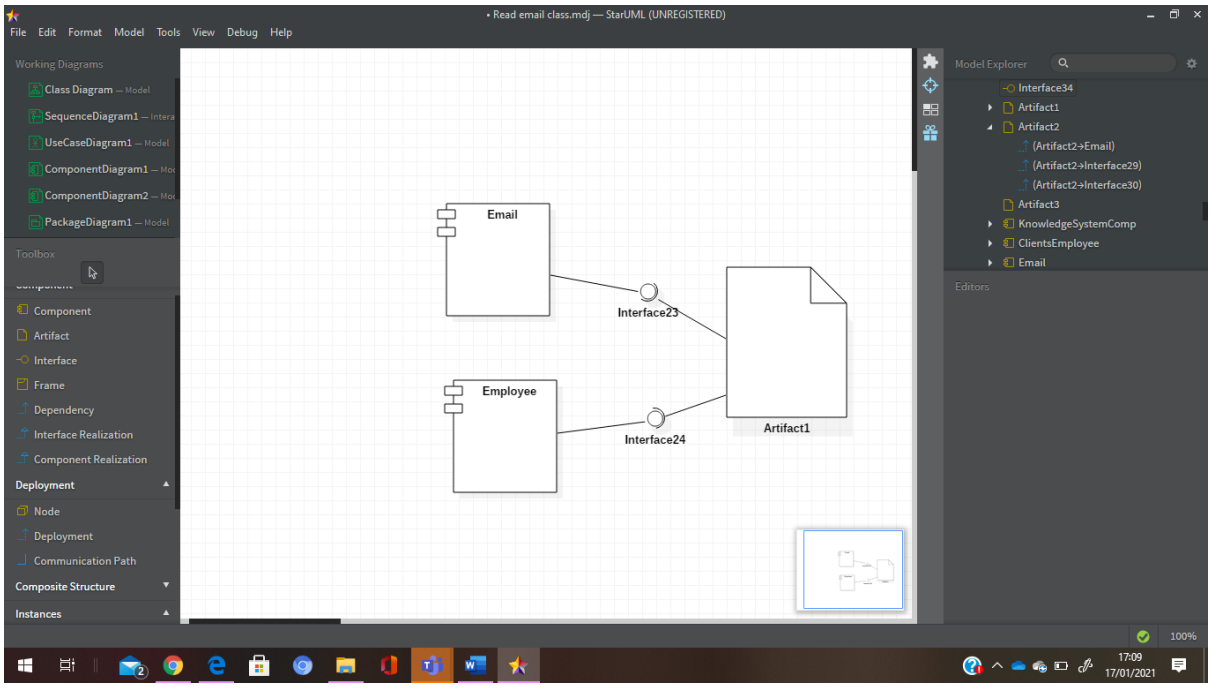


I start by dragging and dropping the two components and artefacts. The artefact is basically the glue code that glues the components together.

I make another component here that has its interface for providing and interface for requiring together. The interfaces that are ready to give services to other components and those that state the services that are needed by other components.

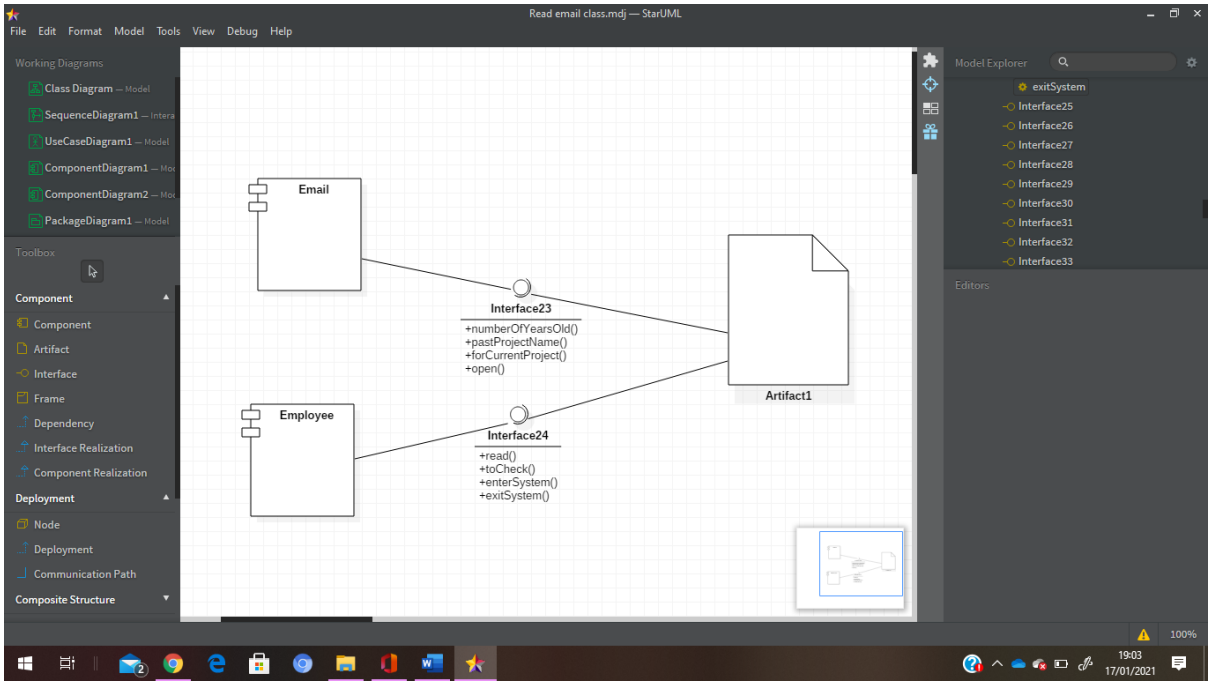


This is like the sequential composition with the adaptor. Email and Employee component being put together to communicate with each other. This makes it a bit like the additive. The artefact stands for the added code that we put in the components to communicate with each other. The code added in the components is the code put in the application to program it. So adding the code is worth it.

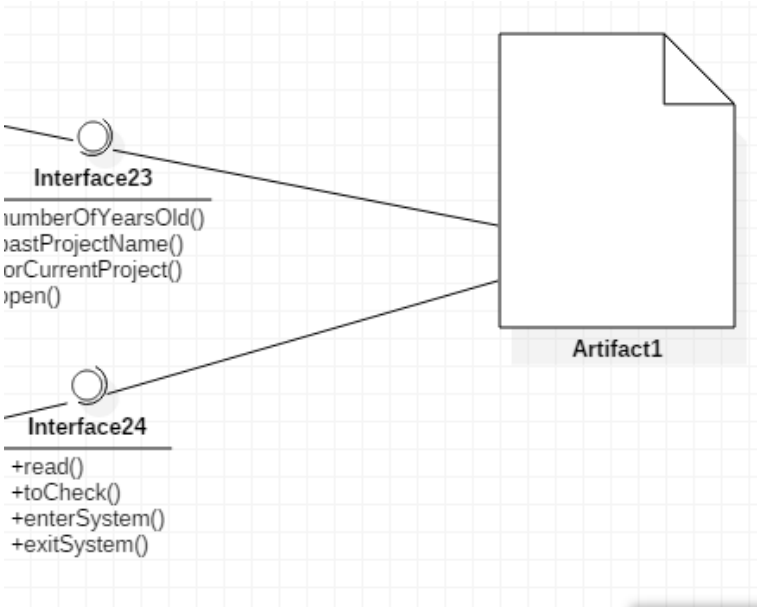


I cannot give the artifact an interface that provides. Only a dependency. I am not able to join Email and Employee to the glue code box on the left with the interfaces that require and provide. It is meant to be the glue code box being joined to the Email and Employee with its provide interfaces, and their require interfaces i.e. dependancies.

If the glue code box on the left was put together with the two components it would give services that are made to work by the code. Code that makes the emails available to be read, stored and forwarded. Plus other data to be passed from the Email component to the Employee as they are glued together.



Then there is the code from the boxes on either side that makes the Employee able to open and work on the applications on the user interface in front of them on the screen. It lets them check and send information. The glue code interface, on both sides, have methods for the code to basically run the applications the employee is working on to do this. I am meant to state the interfaces with the methods for the glue code boxes but am not able to do so here. The methods of Employee and Email are a clue of what they could be.



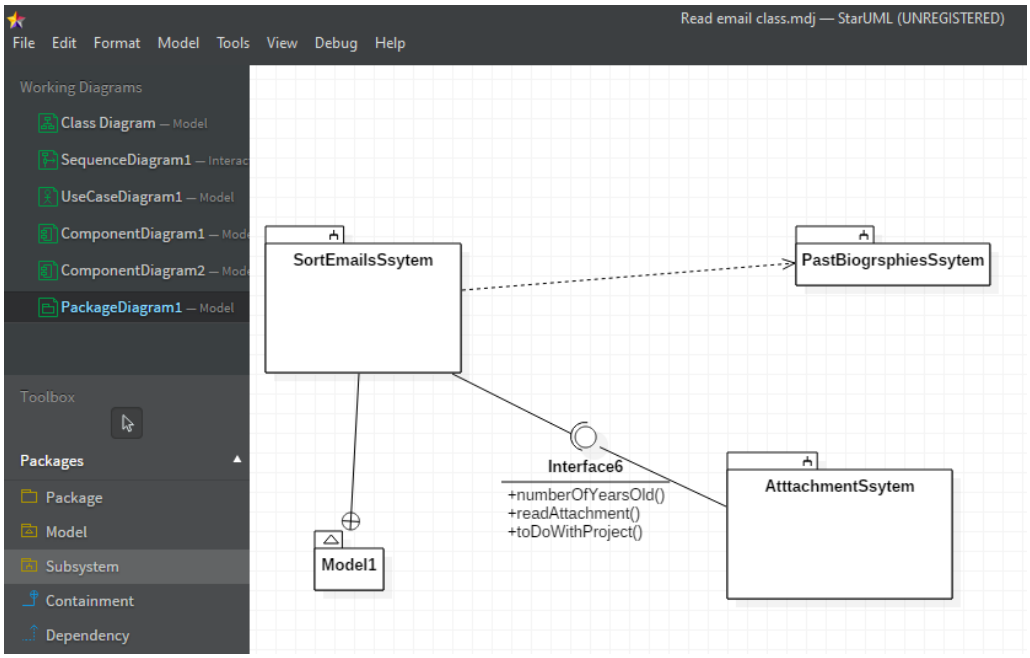
There are the dependencies of the glue code box joined to the provide interfaces of Email and Employee. This is the outside part of the two components the dependencies are joined to for the code to make the services be given by the components. It allows the box to communicate with Email and Employee.

Task 4

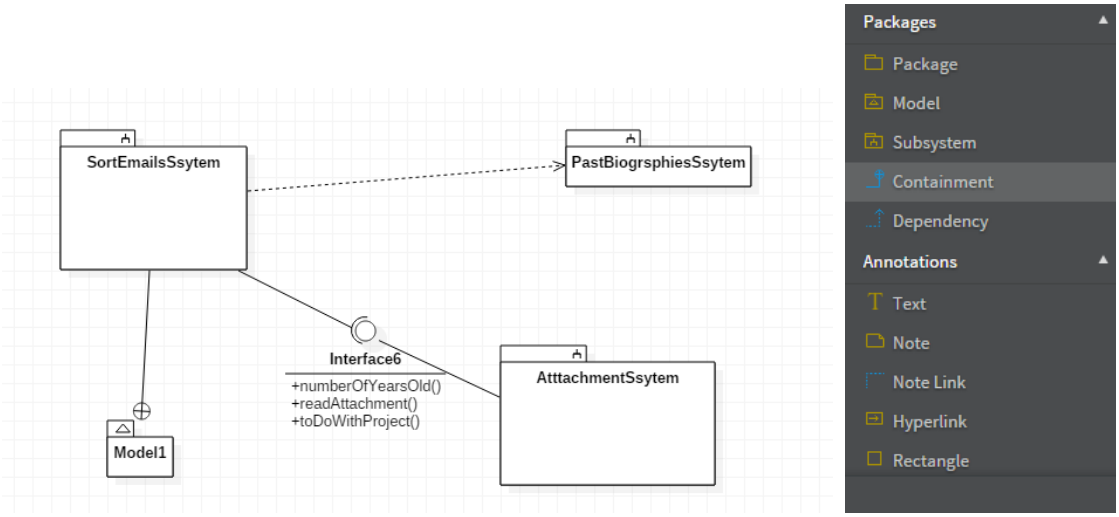
Architectural design

This stage focuses on the relationships between subsystems and models. A sub-System is a system having methods that are run on their own rather than with services the other sub-Systems give. The subsystems have interfaces but the components, that are in the subsystems, are not shown here. Subsystems are linked by a realisation in the models. The diagram below shows all this:

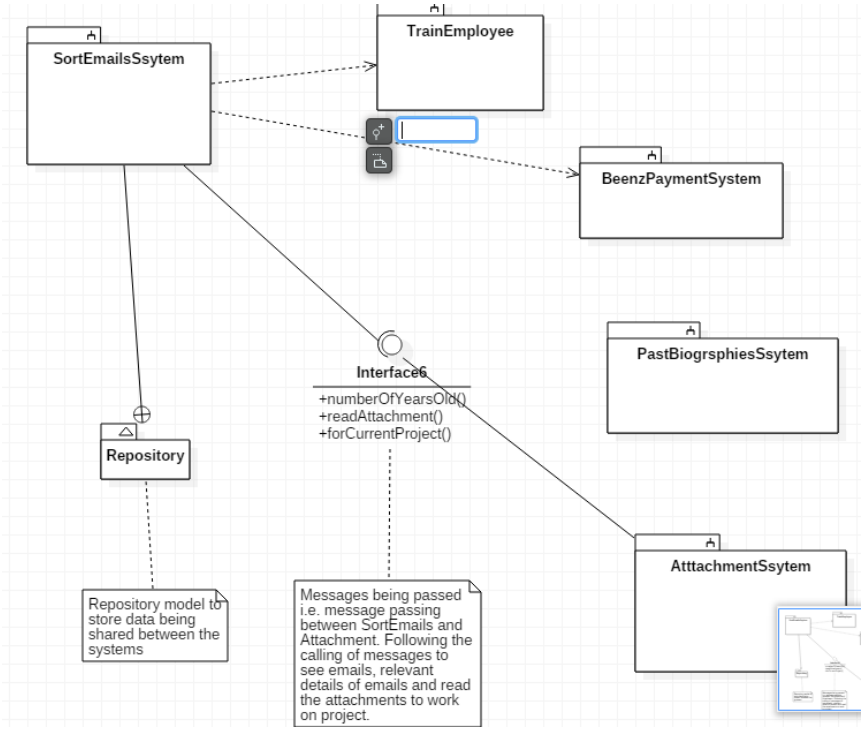
Diagram



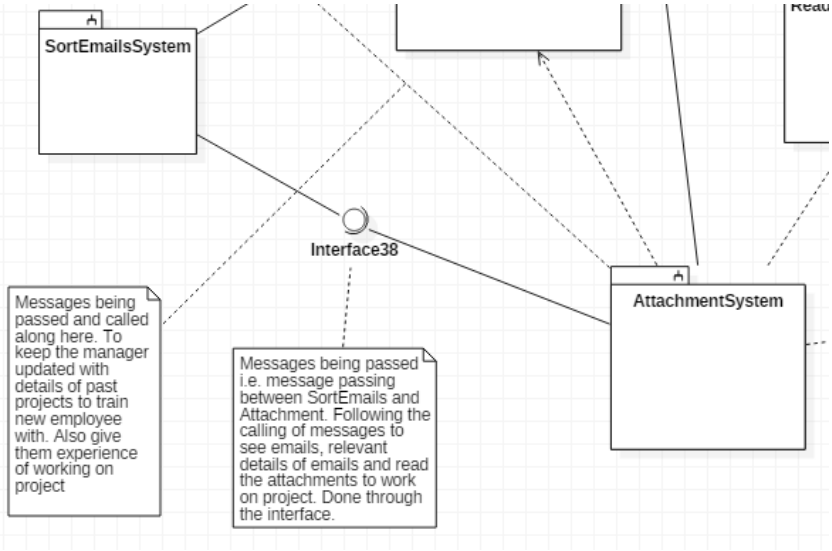
The susbsystems are linked together by dependancies and provided and required interfaces. My read email use case I have worked on in assignment 1 counts as a subsystem. I have put it as **Attachmen**t from here since that is the class in my use case. I decide to change it to **Read Emails** towards the end of this task. It links to other subsystems to do with the case study. These are PastBiographies as in reading biographies of past employees for help and SortEmails as in putting the emails in order. We are not showing the internal components of them here. We keep the systems as they are.



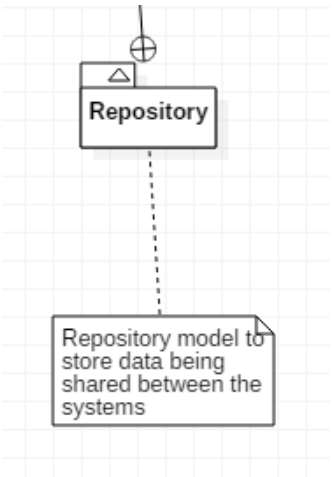
Subsystem SortEmailsSystem has its interface. It has a dependancy as a link to PastBiographies subsystem. It has a containment linking it to the model. It has the required interface connecting it to the provided interface of Attachment. These two systems are connected together. Attachment has methods that give a service through the interface like the components do.



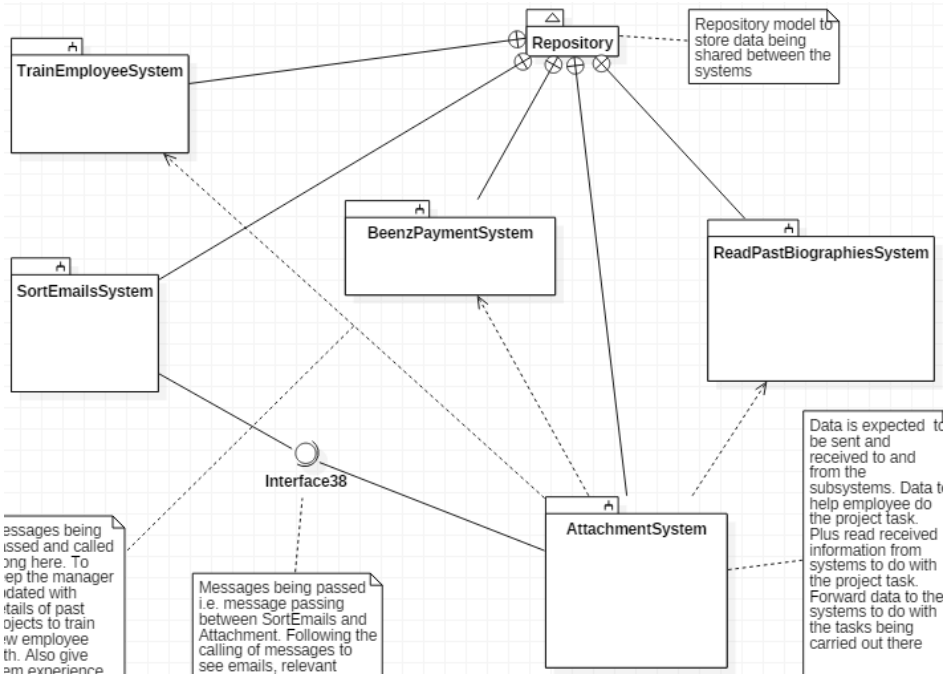
Between these two systems there is the information that is needed for there to be methods i.e. operations. It is not worth having one subsystem call an openAttachment() method. The information that is needed with the two is likely to be the numbers of years ago the email was sent. The project it is do with is is the information that is needed too.



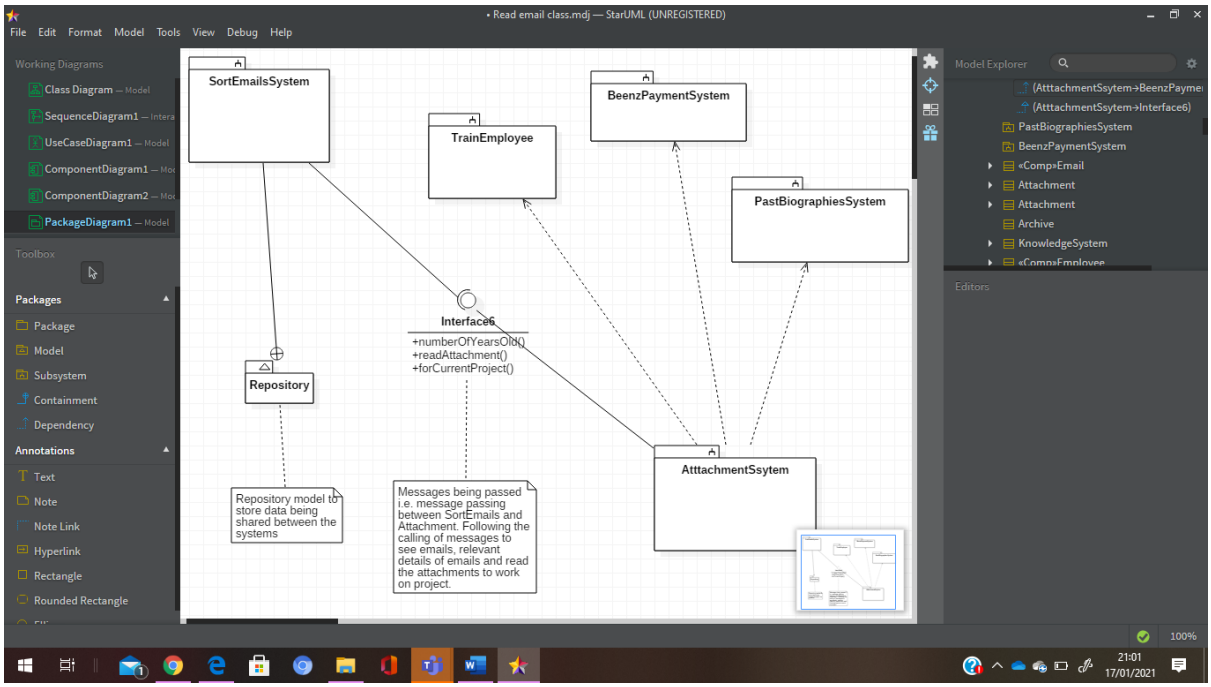
Any communications of messages between the subsystems are based on the type of information that is needed to do different tasks. Different tasks that focus on the project since this is communication between the systems. The methods in the interface are turned into messages to be called and passed between systems.



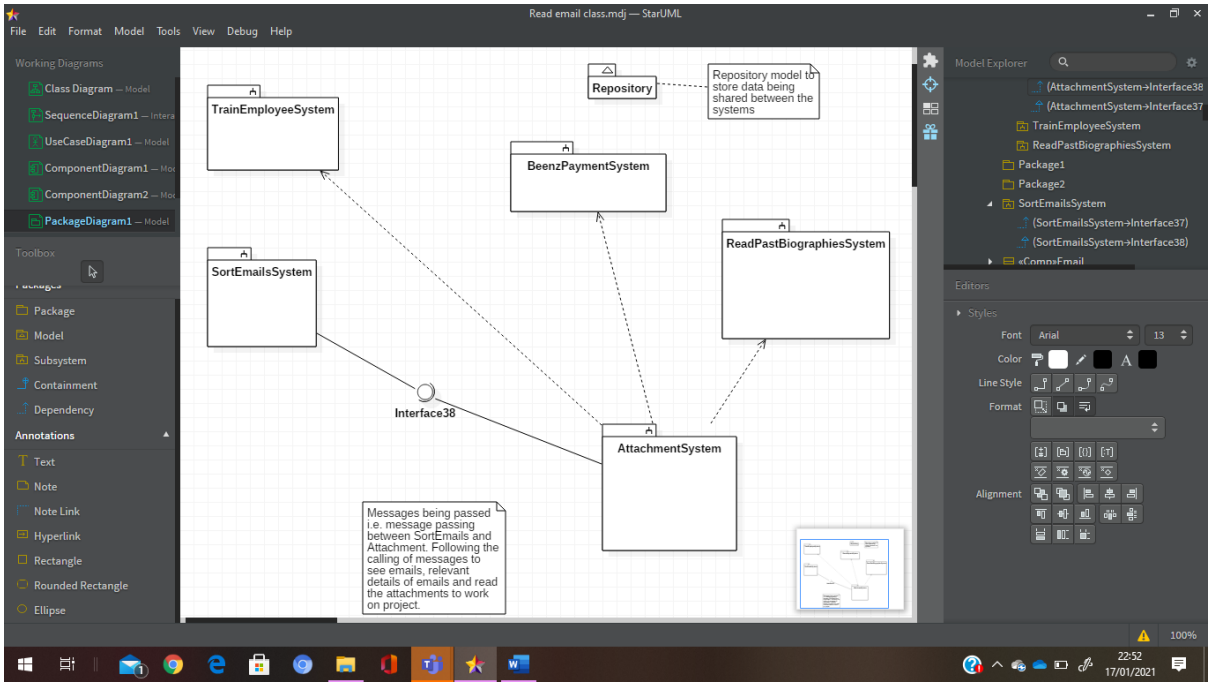
The model in this system is a repository model to help the susbsystems when they divide and pass a big number of data to each other. It stores the data being divided and passed by the systems. Every subsystem can get this data.



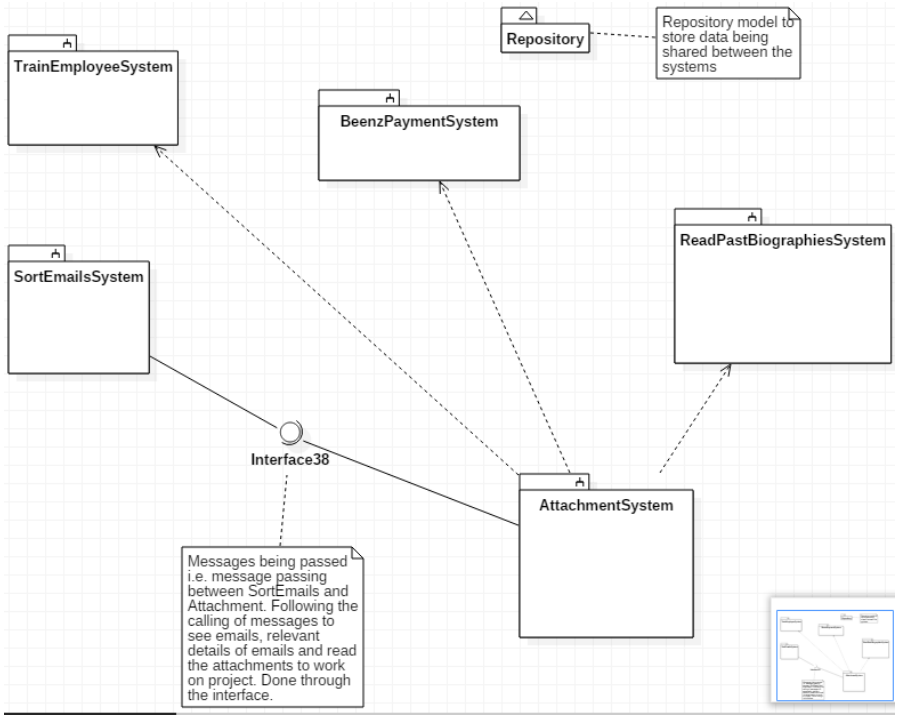
By now the systems are connected to the repository as containments. This tells us it is the that allows data being divided and passed between the systems to be put in the repository. When the sub systems share data they do so by having their own database to check and fix.



The PastBiographies system covers for employess reading biographies of employees who have left the firm. As the firms focus of the work is on the project employees go through biographies to find past employees with suitable experience to help them with their project. Therefore this system is part of this whole system as a dependancy of one of the subsystems being SortEmails.



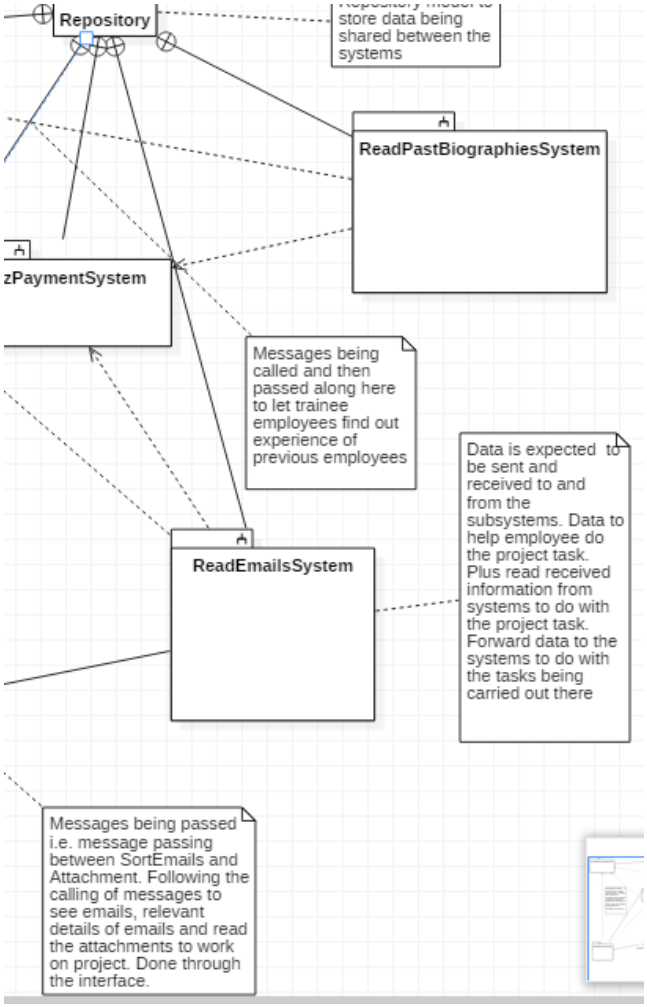
The cohesion of my design is that my ReadEmail Sub-system has classes that are not completely fixed on what they are exactly meant to be doing. The classes do different types of tasks on the right track of what they are meant to do. The same with dividing and passing data plus receiving data from other Sub-systems such as dividing, passing and receiving information to and from the manager training employees to work on the project with the help of emails. Not being completely fixed on exactly what it is meant to do but doing it on the right track. The cohesion of the Sub-system is rather low than high thanks to the way its classes and the system itself does the tasks they are meant to do.



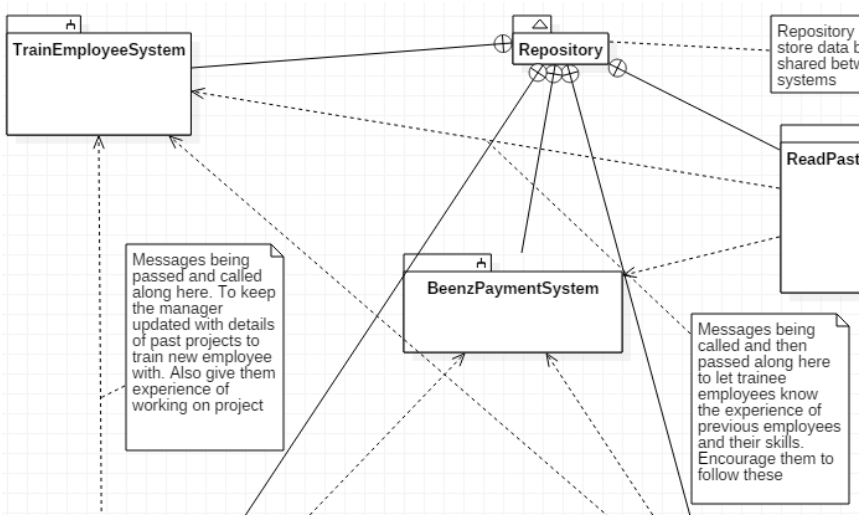
The coupling stage is on the relationships between the classes and the Sub-systems. How they work with each other depending on the use case. Making an attribute of a class or code to run the classes

and the sub-System different does not make the other classes or sub-Sytems they are working with difficult to run. Suppose an attribute in the Project class or code to run it was changed. Doing so does not bring on problems with the Attachment class it is to do with. The details in the attachments such as who completed a past project are still there to help with doing the current project. This means the data shared by the system with other Sub-systems it depends on such as SortEmailsare still the same. Data on the email being in order the attachment is in stays the same. This is a low cohesion.

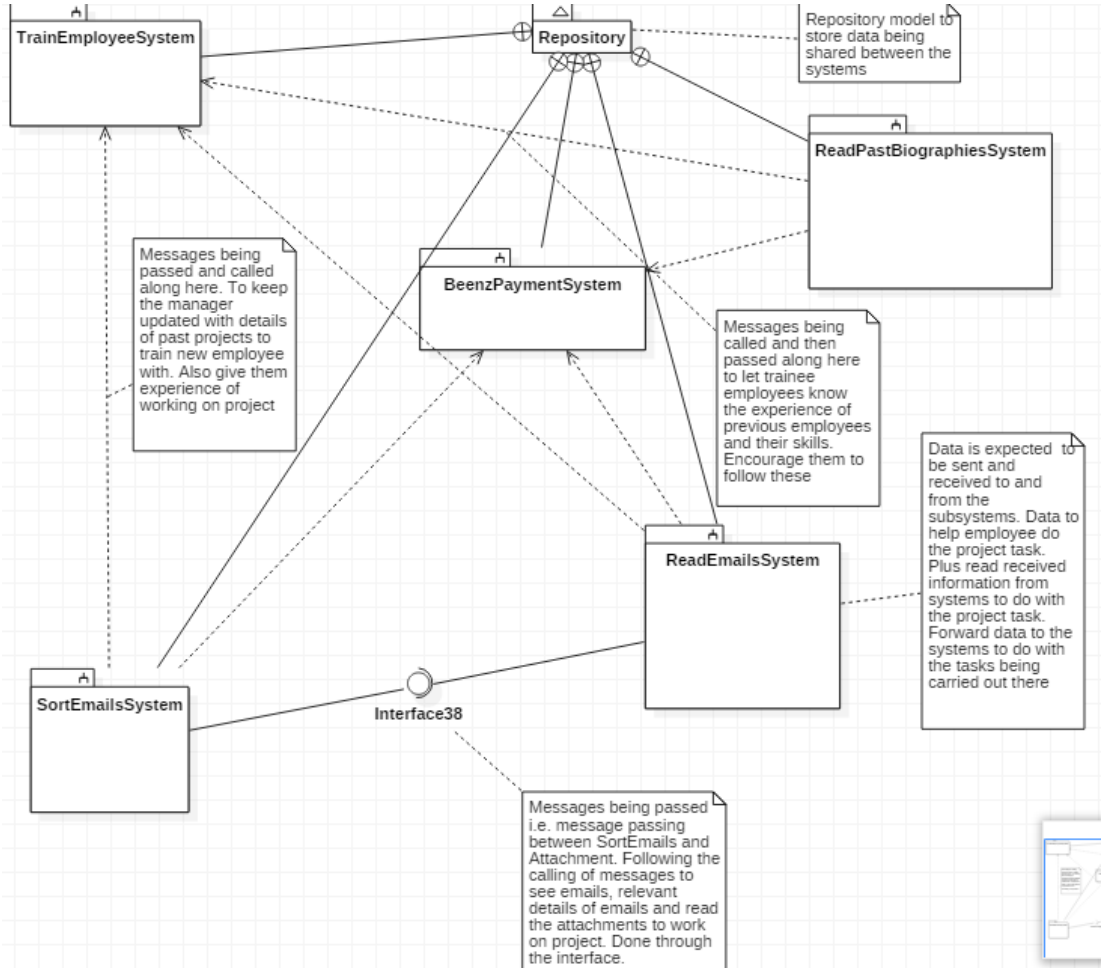
SortEmails Sub-system is connected to TrainEmployee Sub-system. A description of the calling and then passing of messages is given in the diagram.



I have decided to change **Attachment** to **ReadEmails**. Description of the process of data being passed along the systems. Especially passed to and from the ReadEmails for employess to send and receive information to do with compketing the project task. This Includes employees who have left the firm having worked on their projects in the past. The BeenzPayment system shares data to reward workers for doing their job with the information in the system.



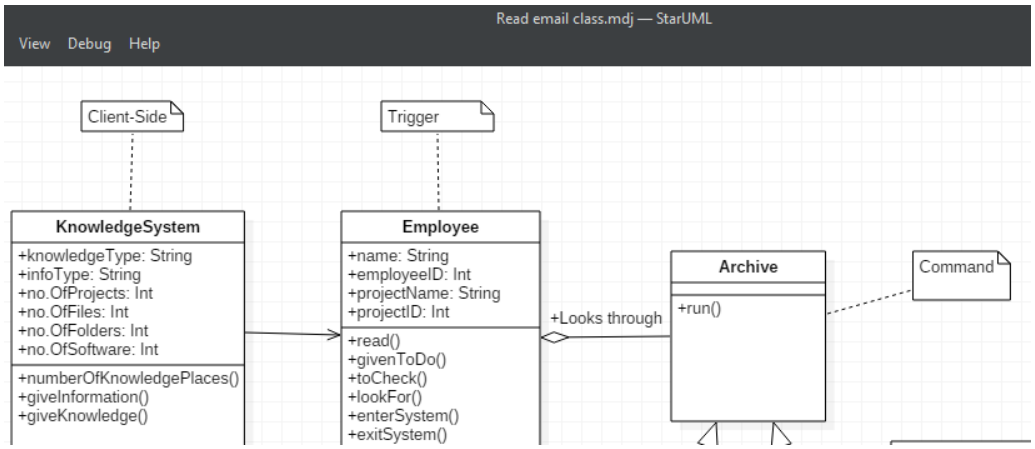
Reading biographies of employees who have left the firm to gain experience and knowledge is vital for new employees when they are being trained. This is by message calls followed by message passing. Given in the description in the diagram above.



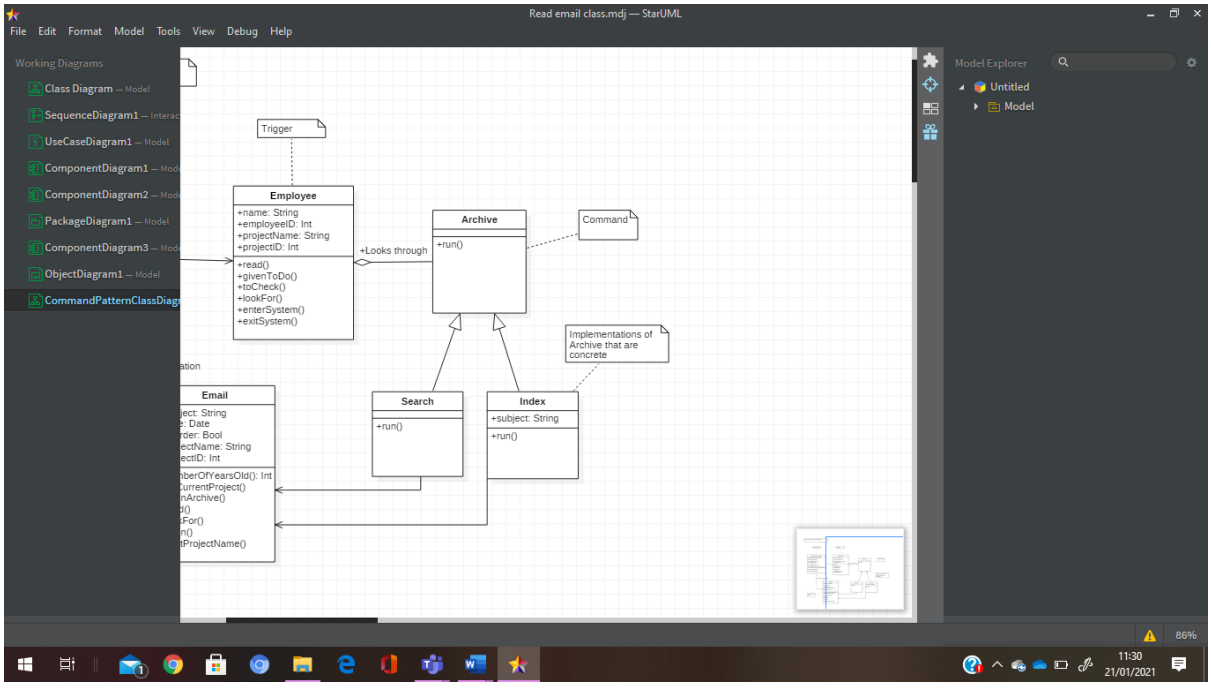
The system as a whole based on the H&K case study. Connections between each Sub-subsystem are shown by dependancies – arrowed lines. Communication through an interface for the information that is needed in the system is shown between two Sub-systems.

Task 5

There are transactions being carried out in the use case. Therefore a command pattern has a solution. A solution that shows this happening. This is done as a class diagram with methods, classes from inside a class (aggregation), arrows showing the implementation and lines to show the methods being called.

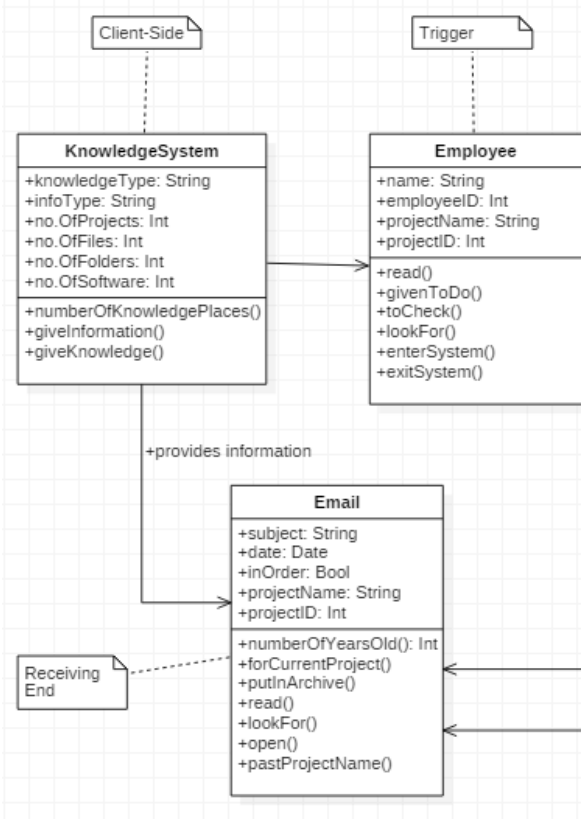


As a starter I have put classes from my class diagram to model the transactions being carried out in my read emails use case . The Knowledge system being the Client, Employee is the invoker and Email being the command.

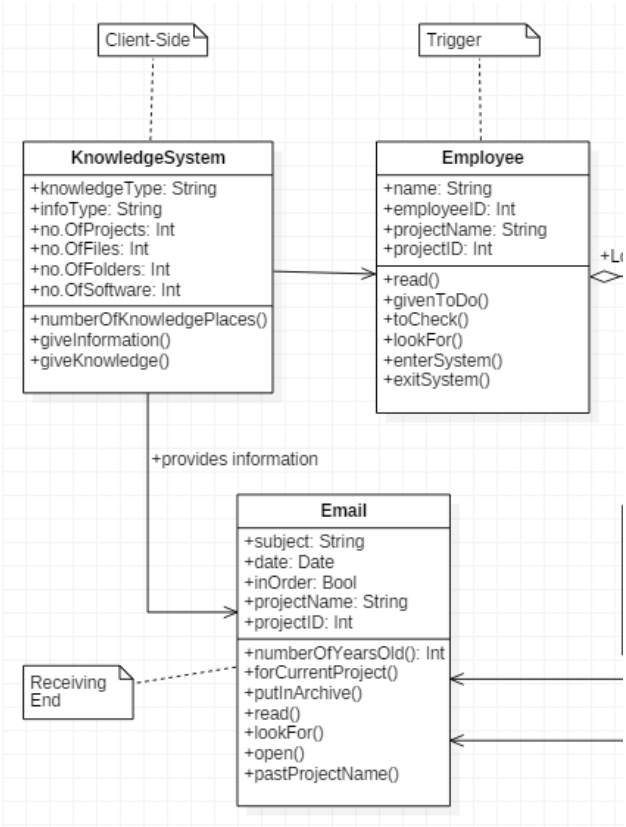


To make the pattern sensible I decide to put Email as the receiver and let Archive be the command. The classes that implement the Acrchive, being the command, are Search and Index. The Search class I have added is for when the employee searhes for the email to read. The classes being implementations do the same as the Archive does which is execute i.e. run() as I have put it. They

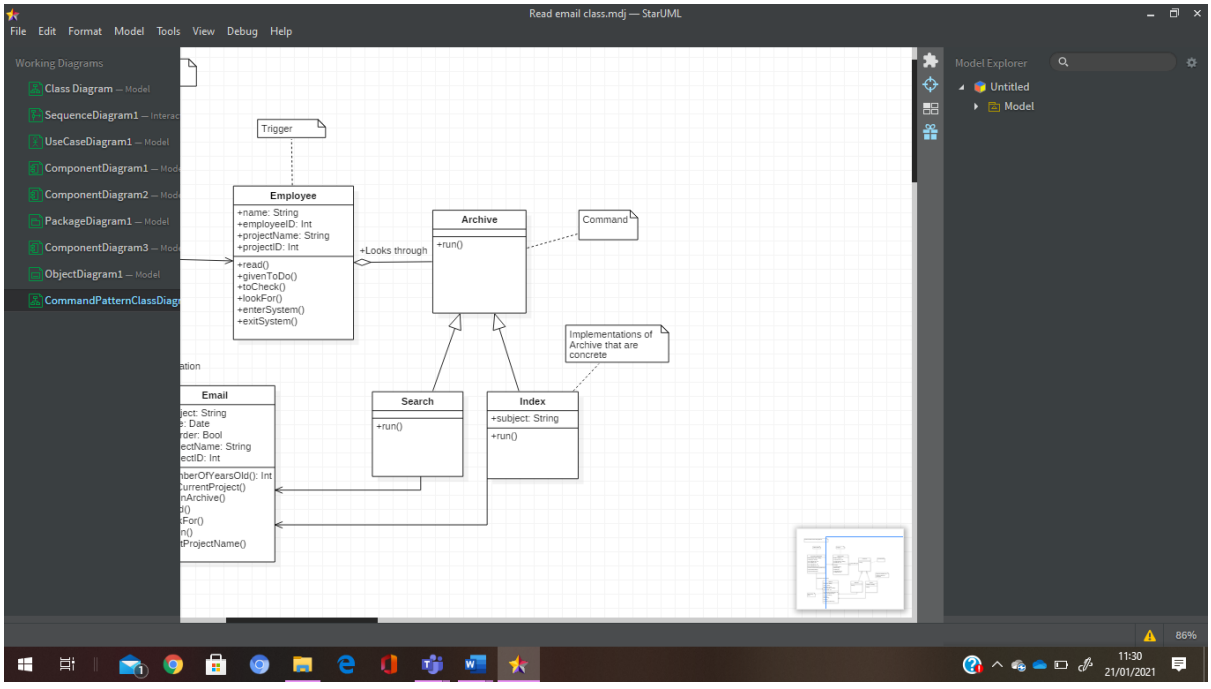
are the concrete commands that put the Archive command into running i.e. executio buy their run methods.



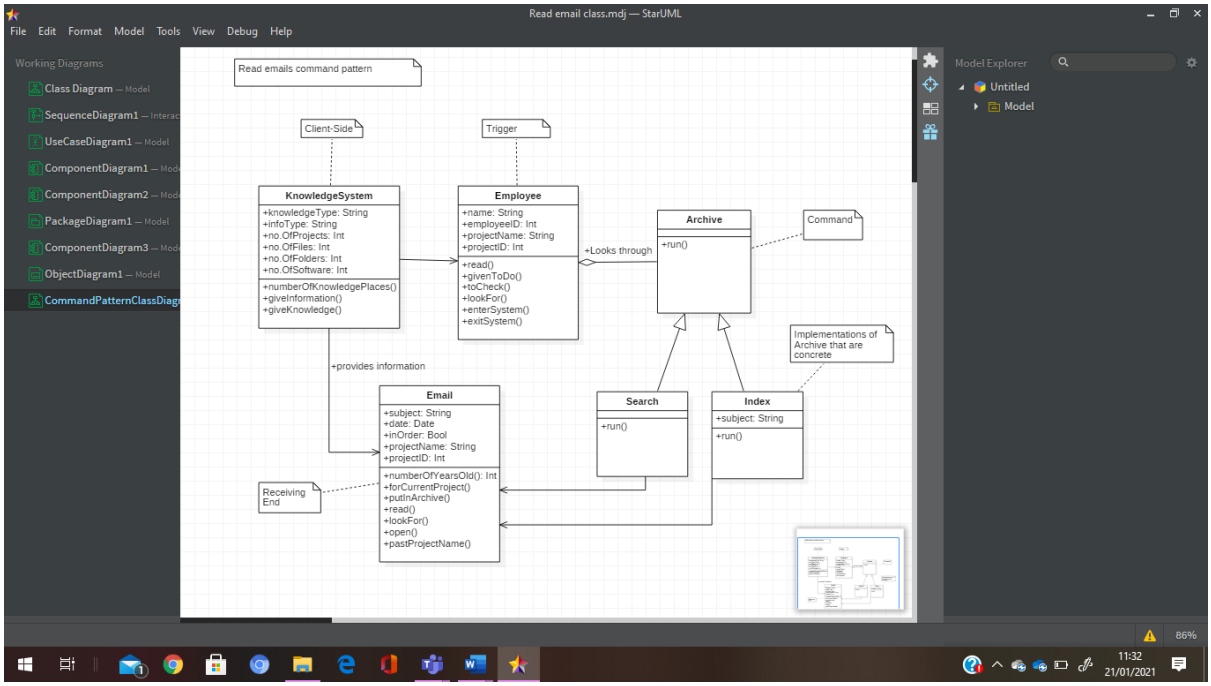
If the employee is looking for emails of past projects that are closely related to the one they are currently working on and to take these actions. So these actions being together. He or she wants to look for the email, open it and its attachments, check how old it is. This means Email is the end that receives the command. It receives the actions that are going to be carried out.



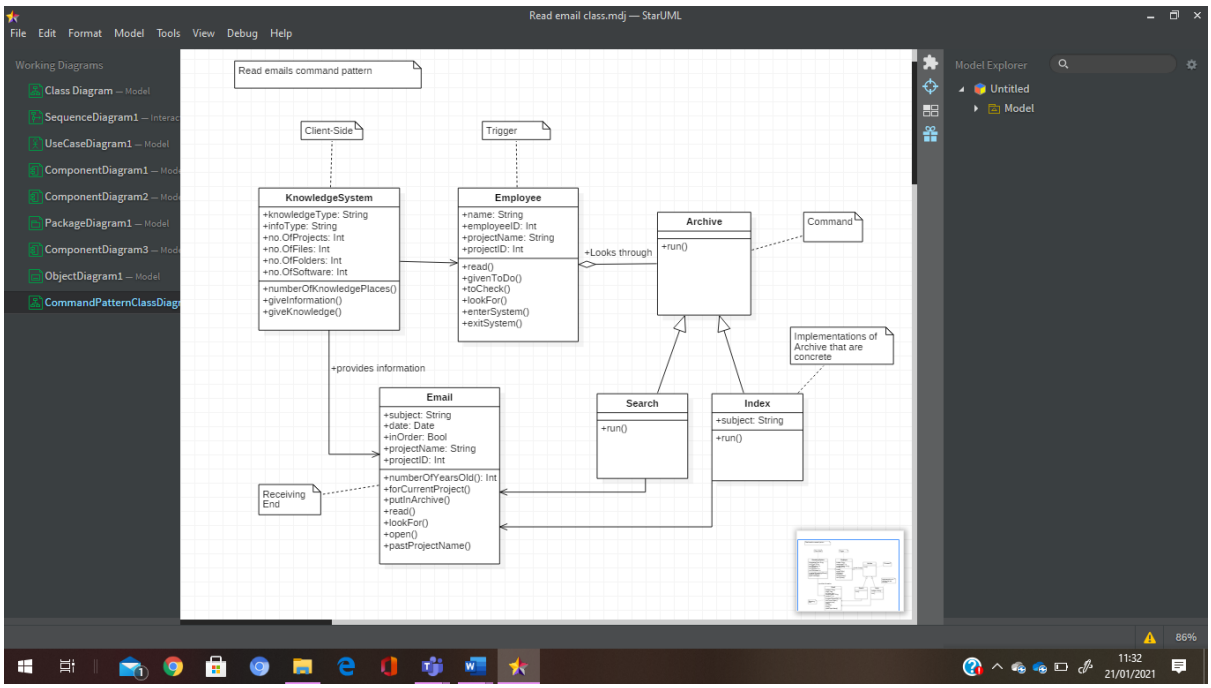
A command to check, look for, open and read. Since we have the knowledge system there is the action requestor in it. This is the invoker also known as the trigger which is being made. The system goes to the Employee class to carry out the tasks lookFor, read, open and numberOfYearsOld . These being the actions that are put on an object. It is not necessary for it to know the class that triggers.



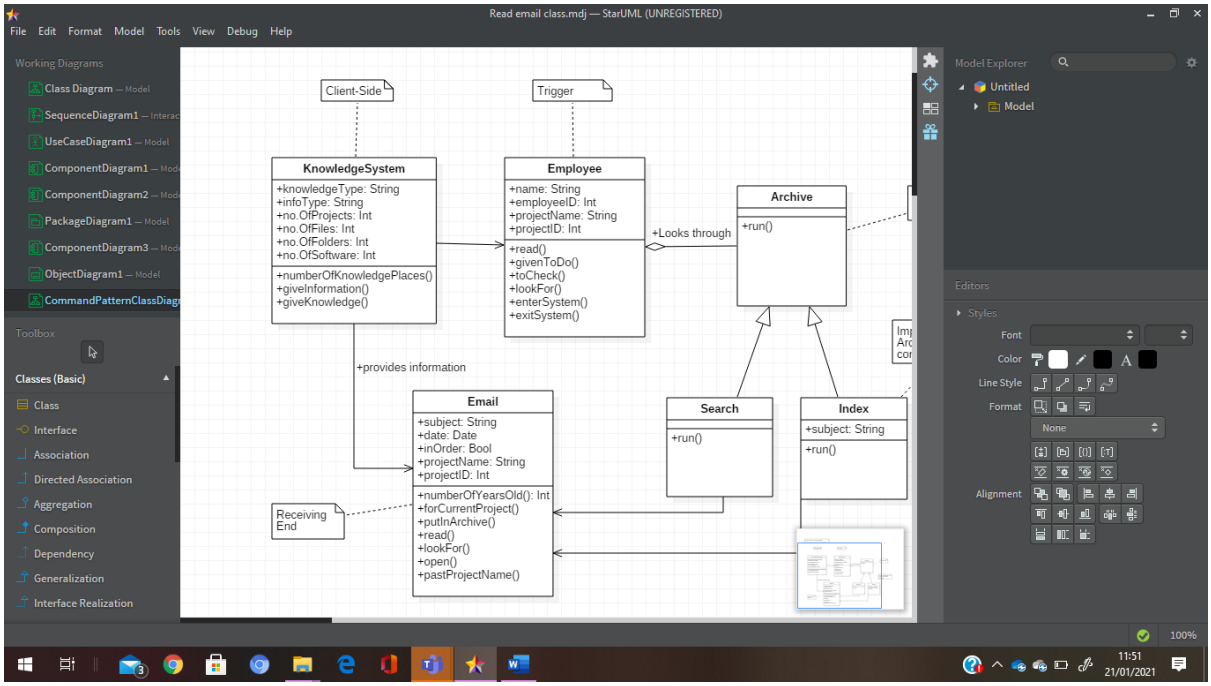
In the diagram at the command all the archive does is call the method in Email to look for, check, read or open. Basically we are placing the actions on an object. The class that needs that it does not need to be aware of. The execute method I've put as run is called by the Archive.



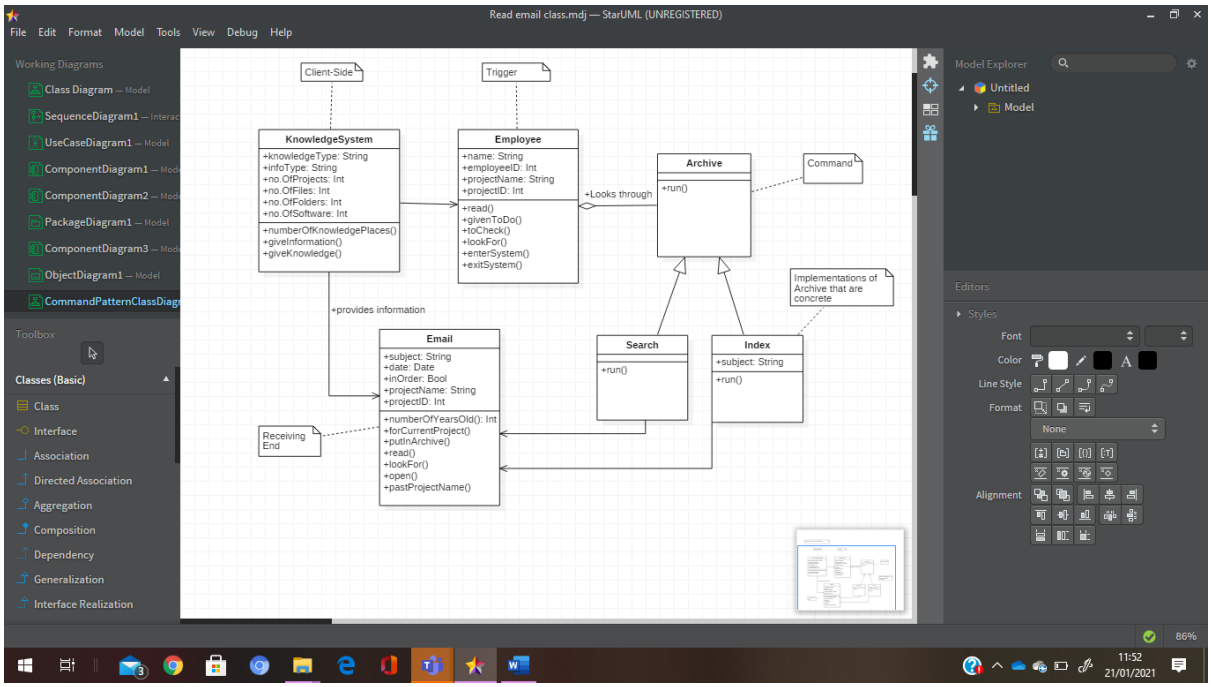
The point of having this command pattern is based on the kowledge system that is to take these actions in the Email. Lets have execute as the only action rather action requesting i.e. invoke the methods in Email.



This only doing i.e. action, based on the way it has been put on the object, goes ahead to trigger that doing whichever one of the Email methods it is. Basically there is another request of the action. This is about the way the action is carried out. It comes to the point of having the pattern being to tell the way the actions move along between different classes by carrying out a command.



If the methods become different the Knowledge System is not to trigger a row of actions if we decided trigger them. It does not need any knowledge of the row of actions. Neither does the Employee. Infact the Employee just calls the run method if one of the check age of email, look for, open or read are being carried out. Plus if I want to have a different method I just make a new class. Could do this if I want to carry out two methods at once.



Having this command pattern focuses on running a row of actions, a row of actions that are not all the same. So there is the calling of method i.e.method calling. This is the run method being called by the Knowledge System. The system then calls the read, check, look for and open.

To top it all up the Search and Index are another type of the Archive class as command classes like a generalisation. They start the action by the Archive command. The knowledge triggers the Email

class and the Search and Index call the methods in Email to be carried out. These are the doing i.e.action methods. The actions are put together in a command

The results of the command pattern are:

- There is a pair of objects – one that requests the operation to carry out a task for a reason and the other that is able to carry out the operation. Command separates the pair of objects
- Like composing components you are able to put commands together to communicate with each other
- You can get new commands without needing to add methods, attributes, data types and change names of the classes. You can still let the classes be the same. So getting new commands is not difficult.
- Commands are object that come under the category as first class. They can be changed and made longer.