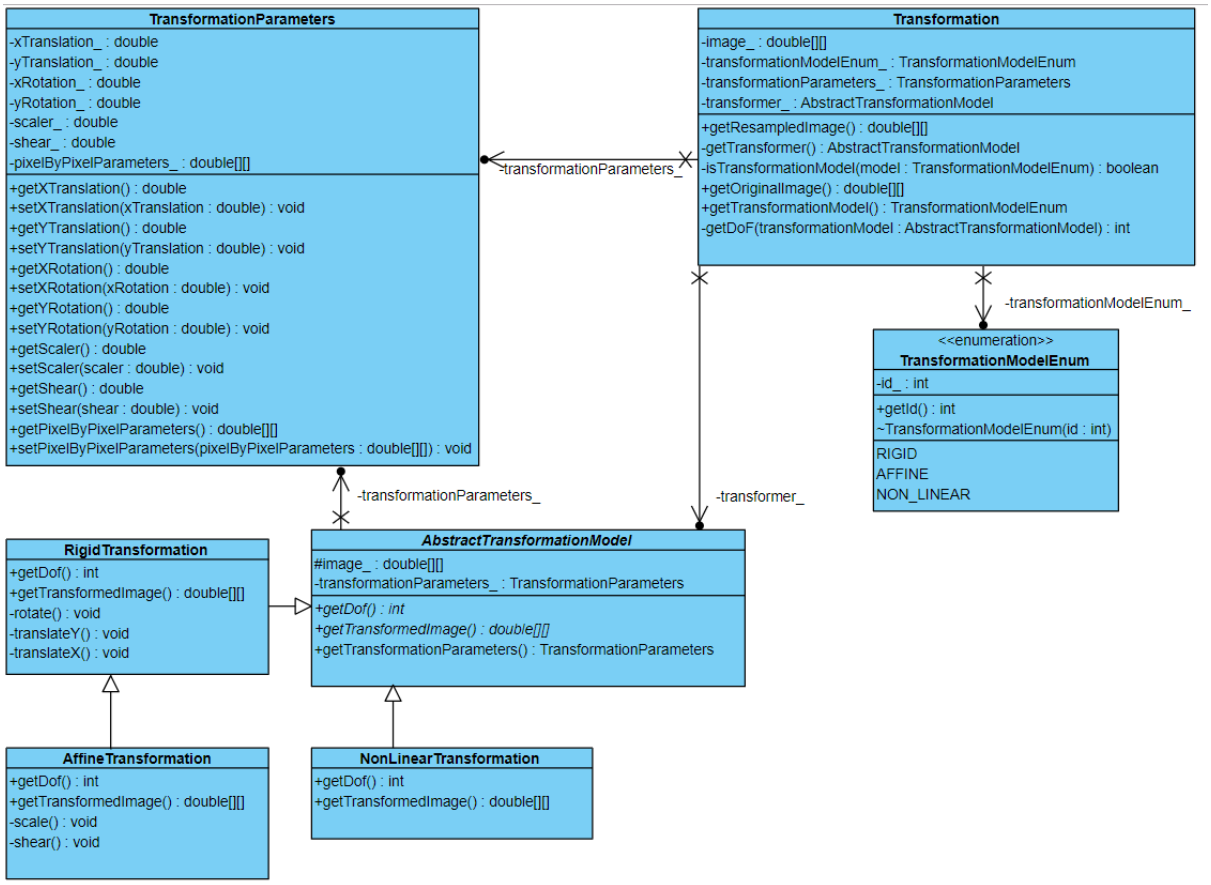
7MR10020: Scientific Programming - Assignment 3

Question 1

For all three blocks, I've chosen to create abstract parent classes for each model/method (**AbstractTransformationModel**, **AbstractSimilarityMethod**, **AbstractDerivativeMethod**) as these will allow implementing more models/methods more easily in the future by providing a sort of "interface" of what is required. Furthermore, in the UMLs I have included some of the inherited methods in the child classes as well to emphasise that that is where the logic is implemented.
I have also omitted adding the cardinality of classes as it does not really offer much in our case. If for example, we had a library system where a certain user could borrow a maximum number of books, then it would make more sense to add it there.

Transformation



1: UML of the transformation block

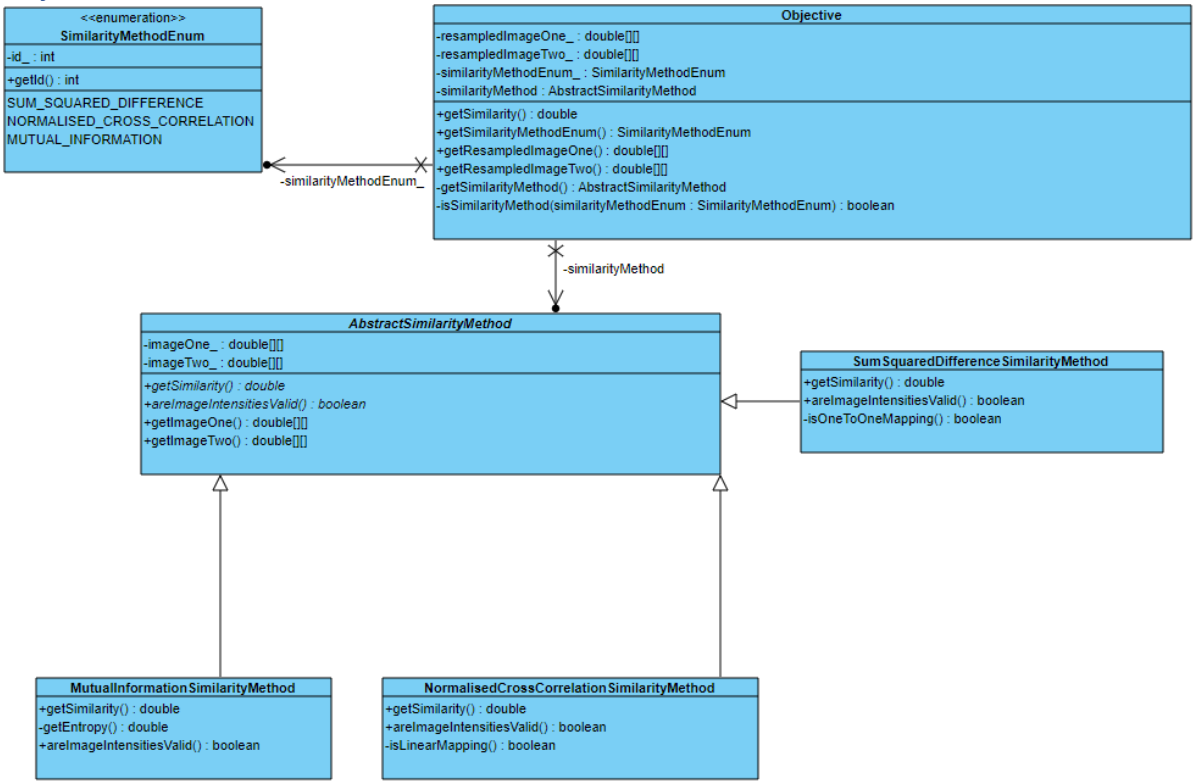
AbstractTransformationModel: The parent class of the transformation model. This class describes the properties and functionality that will be used to transform an image. The classes **RigidTransformation**, **AffineTransformation** and **NonLinearTransformation** inherit from it and provide the actual logic for the transformation, depending on the specifications provided. One thing to note here is that **AffineTransformation** further inherits from **RigidTransformation** as it requires the logic of the rigid transformation. Therefore by inheriting it, it can simply call its **getTransformedImage()** function and further build on it.

Transformation: Serves as a sort of "wrapper" around the transformation model classes, simply requiring from the user to provide an image, the preferred transformation model (through the **TransformationModelEnum** enumeration class) and an instance of the **TransformationParameters** class. Once the transformation is finished, the resampled image can be retrieved through the **getResampledImage()** method.

TransformationModelEnum: This could have simply been a string/id/boolean, but the use of the enum will allow further attributes to be defined if required in the future.

TransformationParameters: A simple wrapper/bean-like object where all the possible parameters can be defined. This is mainly used to avoid having many parameters passed to the classes and their constructors.

Objective



2: UML of the objective block

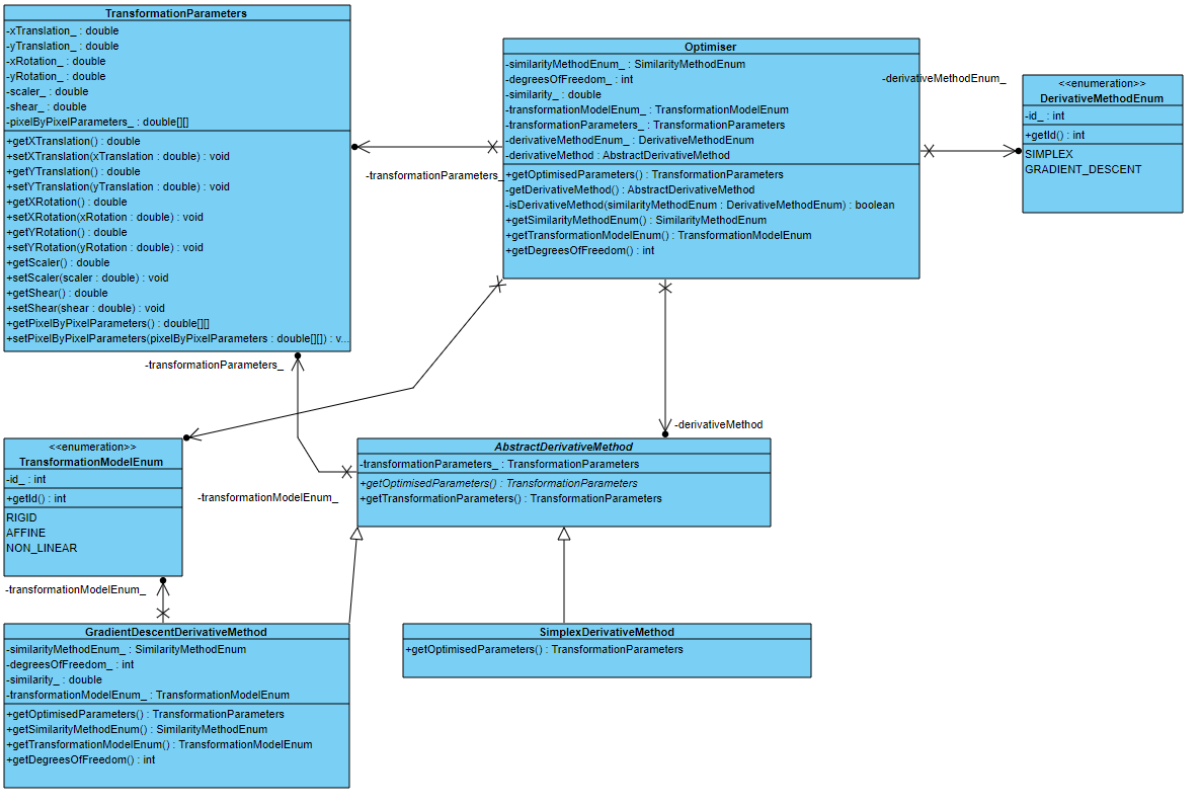
AbstractSimilarityMethod: The parent class of the objective/similarity model. This class describes the properties and functionality that will be used to retrieve the similarity between the two images. The classes **SumSquaredDifferenceSimilarityMethod**, **NormalisedCrossCorrelationSimilarityMethod** and **MutualInformationSimilarityMethod** inherit from it and provide the actual logic for the objective function, depending on the specifications provided. One thing to note here is that while all 3 child classes inherit the **areImageIntensitiesValid()** method, only **SumSquaredDifferenceSimilarityMethod** and **NormalisedCrossCorrelationSimilarityMethod** provide further logic from it through **isOneToOneMapping()** and **isLinearMapping()** respectively.

Objective: Serves as a sort of "wrapper" around the similarity method classes, simply requiring from the user to provide the two resampled images, the preferred similarity method (through the **SimilarityMethodEnum** enumeration class). The objective can be retrieved through the **getSimilarity()** method.

SimilarityMethodEnum: This could have simply been a string/id/boolean, but the use of the enum will allow further attributes to be defined if required in the future.

Here we could have a class called **AbstractObjectiveMethod**, from which **AbstractSimilarityMethod** inherits to allow for further scalability in the future but since for the purposes of this exercise it wasn't required, I chose not to include it to simplify the diagram.

Optimiser



3: UML for the optimise block

AbstractDerivativeMethod: The parent class of the derivative/optimiser methods. This class describes the properties and functionality that will be used to optimise the parameters. The classes **SimplexDerivativeMethod**, **GradientDescentDerivativeMethod** inherit from it and provide the actual logic for the optimisation depending on the specifications provided through the **getOptimisedParameters()** method.

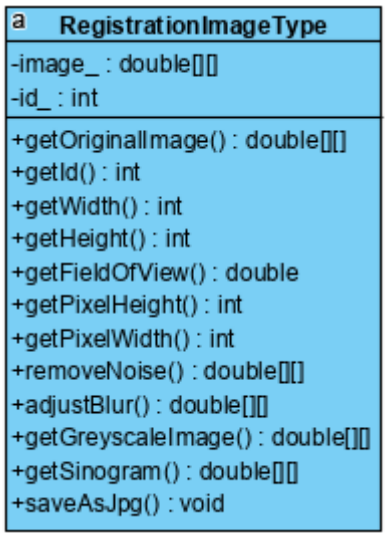
Optimiser: Serves as a sort of "wrapper" around the derivative method classes, simply requiring from the user to provide the transformation parameters, the preferred optimisation method (through the **DerivativeMethodEnum** enumeration class) and an instance of the **TransformationParameters** class. It doesn't contain any real logic, other deciding which optimisation model to instantiate. The optimised parameters can be retrieved through the **getOptimisedParameters()** method.

DerivativeMethodEnum: This could have simply been a string/id/boolean, but the use of the enum will allow further attributes to be defined if required in the future.

TransformationParameters: A simple wrapper/bean-like object where all the possible parameters can be defined. This is mainly used to avoid having many parameters passed to the classes.

Like in the Objective block, we could have a class called **AbstractOptimisingMethod**, from which **AbstractDerivativeMethod** inherits to allow for further scalability in the future but since for the purposes of this exercise it wasn't required, I chose not to include it to simplify the diagram.

Question 2



While some of the examples here might not be 100% relevant, the idea was to add commonly used methods and utility methods that will lead to cleaner code and help the further development of the system with the least amount of effort.

getOriginalImage(): Should we ever need to use the original image before any of the pre-processing steps were done.

getId(): If the image can be found in a database, having its ID might useful to query data that might not be available here.

getWidth(), **getHeight()**: Utility methods to avoid having to recalculate the dimensions every time.

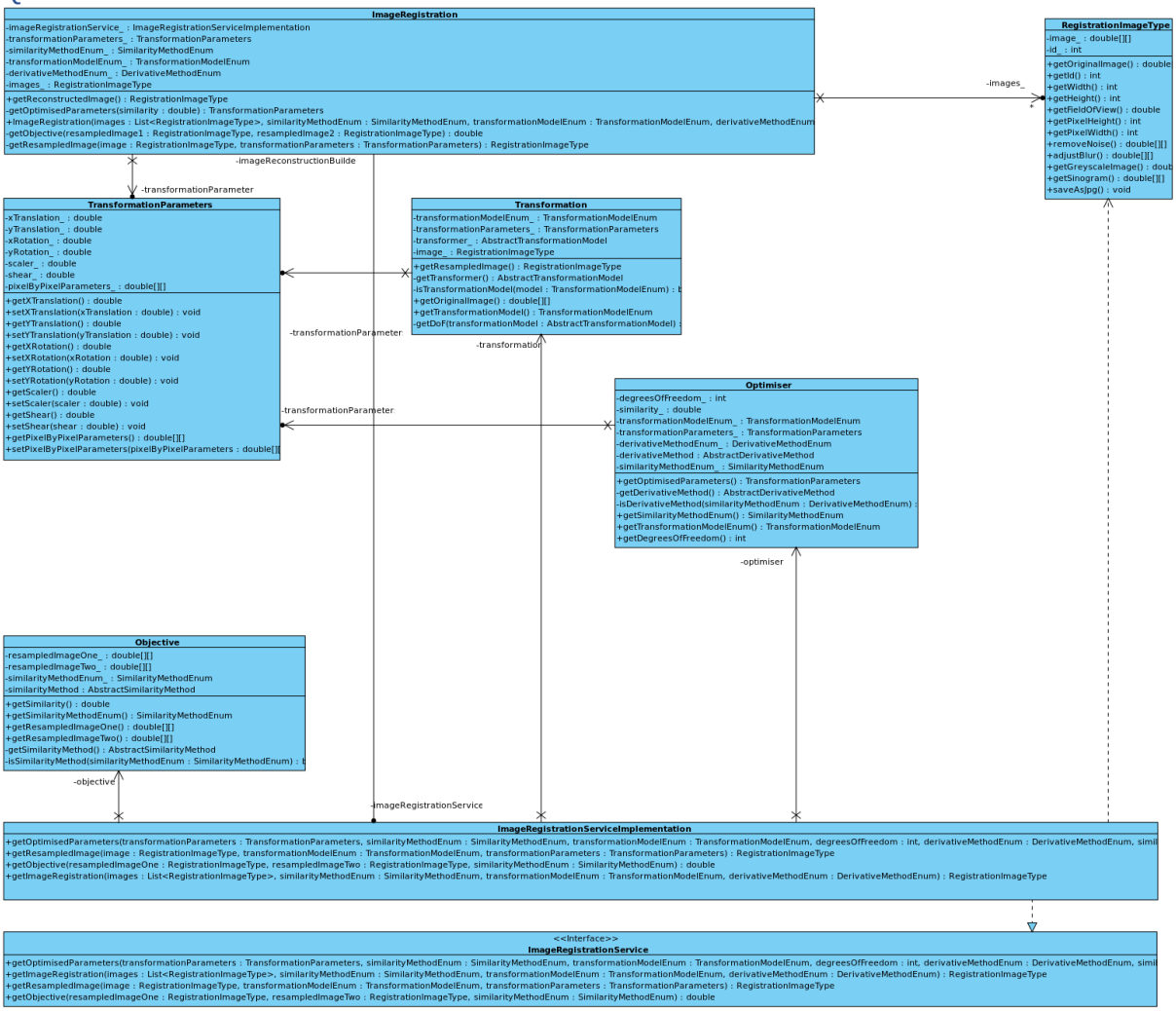
removeNoise(), **getGreyscaleImage(), getSinogram(), adjustBlur():** Some pre-processing steps that might be useful to have. For example, the sinogram is smaller in size than the original image and can be useful for less complex registration. Even if these functionalities are not 100% relevant in this case, the idea is to include such methods to simplify the developer's job of looking for them in the codebase.

getPixelHeight(), **getPixelWIdth()**, **getFieldOfView()**: Useful information to have scaling for example.

saveAsJpg(): Perhaps even more controversial to include this here than the pre-processing methods. The idea is to have an easy way to view the image. Perhaps it should just be a display instead of a save method.

These are just a few examples, we could go even further and add functions for resizing, converting to different types (i.e. array -> list) and so on but that will depend on what we find out is necessary while developing the framework.

Question 3



4:UML of the whole framework (high level)

This diagram shows the high-level implementation of the framework, so I have omitted the low level classes that compose the three blocks.

The class combining the three blocks is the **ImageRegistration** class. In it, is all the logic for utilising the functionality of the three blocks. Instead of calling the three blocks directly from it, I have decided to define an interface, **ImageRegistrationService**, that describes the process of performing image registration. **ImageRegistrationService** contains a call for each block, along with a call to simply get the end result (**getImageRegistration()**). If a developer simply wants to get the image registration, they just supply the appropriate input to **getImageRegistration()** without really having to understand what's going on. If they need to do something more specific with each block, they can still do that.
ImageRegistrationService simply describes the steps for image registration and so **ImageRegistrationServiceImplementation** realises/implements the service. If a different implementation is required (i.e. a completely different objective block), simply creating a new realisation of **ImageRegistrationService** makes this possible.

Furthermore, by not directly linking the three blocks, we can now use them individually should we ever need to in a different implementation of the framework.

Question 4

Whether we are talking about a user interface or an API interface, we must always try to keep it as simple as possible and try to apply standard conventions to make it easier for our users to use.

User Interface (UI)

To start designing the UI, we must first try to figure out what kind of users will try to use our tool. For this exercise, I will assume that the most common user is a scientist/researcher, with a decent experience in using software but with a busy schedule and no time to figure out what's wrong unless told to by the tool.

Simplicity: As mentioned before, the goal is to keep the interface as simple as possible by avoiding unnecessary elements (such as buttons and huge blocks of text that no one will read).

Consistency: Using common UI elements (checkboxes, input boxes, etc.) helps the users feel more comfortable, especially when using a tool for the first time. A simple example of this is the exit button that is (or should) always be a button on the top left or right corner of the screen, depending on the operating system. Furthermore, when establishing a rule in our UI the aim should be to try to maintain that rule as much as possible. For example, if we teach our users that information can be found as tooltips when hovering over a button, the aim should be to always display information in that manner.

Design: A great amount of time needs to be spent on the design and order of things. A hierarchy of how options (such as which transformation model) should be selected should be established. Hiding options (for example by using dropdown lists) is a common practice, but if only a few options are available (the three transformation models), we should consider simply displaying all of them to avoid unnecessary steps/clicks.

Defaults: Building on the previous point, a common problem with advanced tools is how overwhelming they can be to use the first time or even the nth time. Users will often want to do something simple/common, so we should consider adding default setups. For example, if the most common use of our tool is applying rigid transformation along with a mutual information similarity, we should consider creating a template of those options that is easy for the user to select. At the same time, adding the option for users to create their own templates for their most common tasks is usually helpful.

Communication: A common problem with tools is the lack of communication between the user and the tool. Clear messages need to be displayed (using language the user understands; not language the developer understands) if something has gone wrong or if something is happening. Moreover, the progress of the task should be displayed in the interface as well. This can be done using as "simple" elements such as progress bars or going as far as iteratively updating images and parameters (i.e. the transformation parameters).

API Interface

Without going into the same amount of detail, the same concepts apply when designing an API as well. In terms of simplicity, the we should avoid adding unnecessary functionality and hiding functionality that doesn't need to be visible. Consistency wise, established naming conventions should be followed (i.e. always use either camelCase or snake_case, not both). Defaults for input/parameters should also be set if applicable and we should even consider hiding some of them through overloading/multiple constructors (although this is not always a good idea). Lastly, regarding communication, we should always return clear and descriptive messages when something goes the wrong (or right) way.