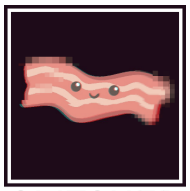


Nizer

Image organization application

A document containing a generic and high-level overview of the image organization application. It describes the key features of the application, such as the functionality It will have, and how it will work.

Authors:	Dawid Grobert
	Julia Boczkowska
Created:	October 21, 2021
Modified:	January 20, 2022
Recipients:	dr hab. inż. Jakub Nalepa
Version:	[1.4]
Classification:	Internal

CONTENT

1 Document History

Version	Date	Author(s)	Description
1.0	21.10.2021	Dawid Grobert, Julia Boczkowska	Initial version of the document that contains the topic description, and a potential solution to the problem.
1.1	05.11.2021	Dawid Grobert, Julia Boczkowska	Improved naming in the sections 2.2.2 and 2.2.3. Added a section related to *Use Case Modelling*.
1.2	01.12.2021	Dawid Grobert, Julia Boczkowska	Added a section related to *Structure modelling*. We have changed the images of the prototypes in section 2.2.1.
1.3	21.12.2021	Dawid Grobert, Julia Boczkowska	Added a section related to *Behavioural modelling*. Updated the Section *4. Structure modelling* with a couple of new methods in some classes.
1.4	20.01.2022	Dawid Grobert, Julia Boczkowska	*Test Planning* section was added to describe testing methods and test cases.

2 Introduction

Nizer is an application that allows to easily organize a huge number of images based on their similarity. In addition to their organization, the fundamental idea of the application will be the ability to view as well as briefly annotate the image.

2.1 Description of the problem from the user's perspective

The application assumes that the user has a huge amount of photos, which the user store on any common storage medium. A common problem of such messy storage is finding photos that differ quite inconspicuously. These can be, for example:

- Photos taken after each other to select the better one
- Photos slightly modified:
  - The difference in the applied filter
  - Differently cropped photos
  - Different color saturation, brightness, or contrast

These are quite common problems occurring to a wide range of people such as amateur, occasional and professional photographers, or even the standard user who keeps his or her backup in a single folder. At some point comes the need to select those better shots and remove unnecessary duplicates.

2.2 Our solution

Our solution comes down to providing a complete solution in the form of an desktop, or even in the future maybe a mobile application, in which the user can indicate the folder on a disk space and the application automatically performs a division into *clusters* (see *2.2.4. Project Dictionary*) with similar images.

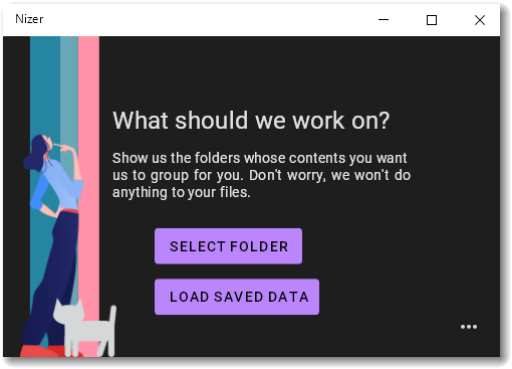
2.2.1 Example application usage

1. The application allows the user to select a folder with photos, whose folders will be searched recursively in search of similar images.
2. Found similar images are grouped into clusters.

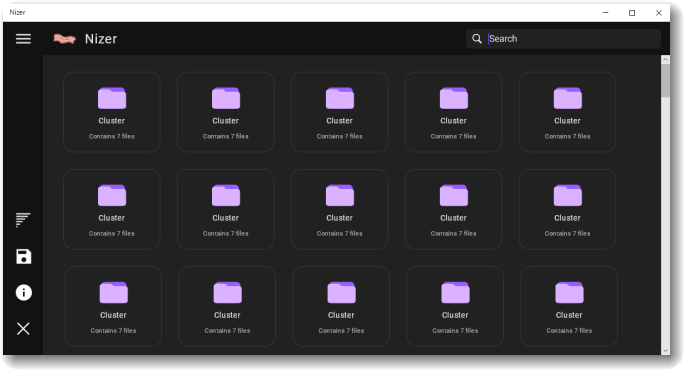
   Warning: No interference with user files takes place at this stage. Their layout and contents remain the same. We avoid a situation that could potentially upset the user - that is, doing something without their knowledge with their data.

3. The user can view the created clusters in the application, although the original file layout remains the same. The application does not copy/modify files at this stage either, it simply displays them in the appropriate groups.
4. The user has the option to caption some photos with an appropriate note and the note remains in the app after opening the file in the app again. The user file is not modified here either.
5. The user has the option to save the cluster layout for reusing it later.

   Info: Future support is anticipated for the ability to copy files and save them physically to disk in the current cluster layout.



An example of how the welcome screen might look



An example of what browsing groups arranged in clusters might look like.

2.2.2 Requirements of the product

We distinguish product requirements into two categories. *Functional* ones, which are related to the user and what they are capable of doing, and *non-functional* ones, which are the goals we want to achieve from the technical side of the product.

Non-Functional

- The application should be multiplatform – this means that we target for different operating systems. We will also consider mobile devices when possible.
- The clusters should remain the same after restarting the program and using on the same dataset.

Functional

- Our main goal is to make the application easy to use for as many people as possible. Many things will be done automatically in order not to increase the complexity of usage of the application. All possible options will be explained in simple and easy way, so user will know what to choose and why. Choices (if any) will be mostly optional.
- For the convenience of using the app, the user will be able to save grouped data. The next time the user enters the application, they will be able to open the layout the user has already saved before.

2.2.3 Implementation Assumptions

We intend to use the following to finalize the project:

- **Qt framework** – to create a graphical and multiplatform application.
- **Perceptual hashing** – in order to calculate the fingerprint of each image, which will later be used for similarity checking between images. To do this, we will use one of the existing libraries for this purpose such as for example *imageHash*. There is also an option here to give the user a optional choice between algorithms such as *pHash*, *dHash*, *aHash* (see *2.2.4. Project Dictionary*).

2.2.4 Project Dictionary

For unfamiliar terms and abbreviations that may appear in the document, please refer to *Project Dictionary* – see *Appendices* at *Section A. Project Dictionary*.

3 Use case modelling

This section describes the top-level architecture of Nizer. It does not describe the architecture of components implementing.

3.1 Actor candidates

Actor Candidate	Role Description
User	The user selects data to group into similar clusters. The user can optionally change how the data is grouped. When the data is ready for viewing, the user may or may not: • Display the images • Save the layout of the data to return to it later When displaying an image, the user can annotate it.
Hardware Memory	Hardware memory is the memory from which data needed by the application is read. It can be an external memory (such as USB, external hard drive) or the memory inside the computer (such as SSD, HDD) where the files such as photos, or saved layouts are stored.

3.2 Use case diagram

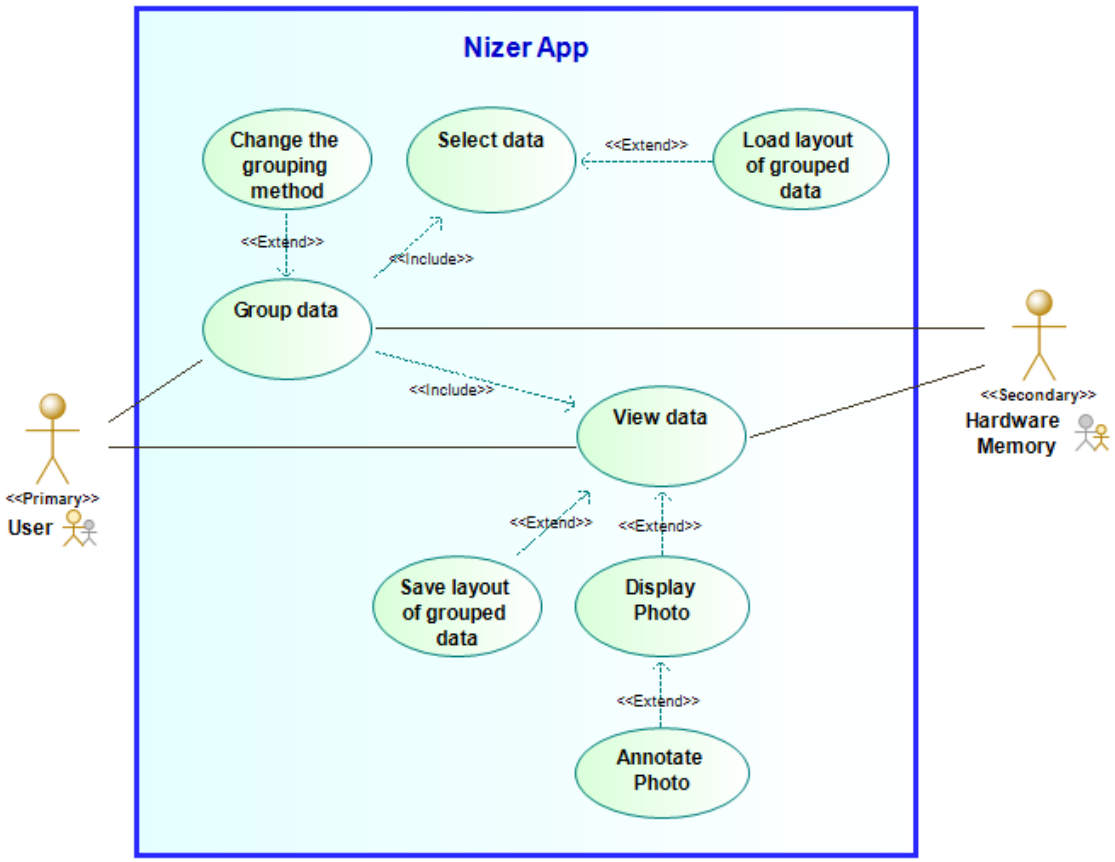


Figure 1: Use case diagram of Nizer application

3.3 Use case example

Use Case Name: Annotate photo and save layout of grouped data

Actors

- User
- Hardware

Basic Use Case Description

1. The user groups the data by selecting the data on the hardware.
2. The user views the prepared data, which is continuously read from the hardware.
3. User displays one of the photos.
4. User annotates the photo with appropriate information.
5. The user then saves the current layout of the grouped data to the hardware.

1. The user can also choose a method for grouping the data.

Alternative flow would be:

1. The user groups data by selecting a previously saved layout of grouped data from the hardware. Then the needed data is read from the hardware based on the layout.
2. The user views the prepared data, which is continuously read from the hardware.
3. User displays one of the photos.
4. User annotates the photo with appropriate information.
5. The user then saves the current layout of the grouped data to the hardware.

4 The user can also override the annotation that comes from the saved layout that was loaded.

4 Structure modelling

4.1 UML class diagram

The model of our application is visualized in the following UML diagram. This is an overall view of the structure of our program. To dispel any doubts the reader may have, we will describe its individual parts in detail on the following pages.

4.2 Simplified class diagram

The simplified class diagram provides a simple and clear view of the application's data structure.

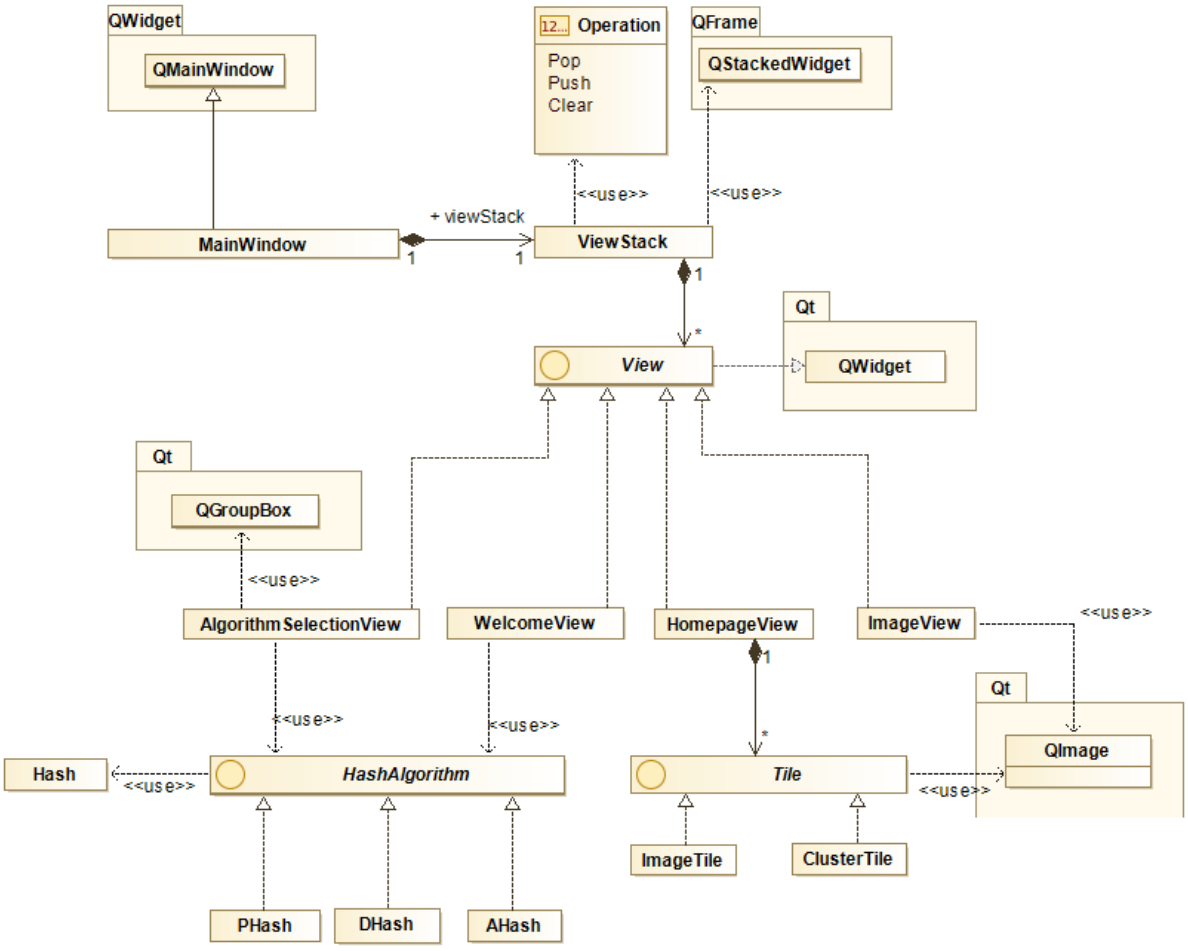
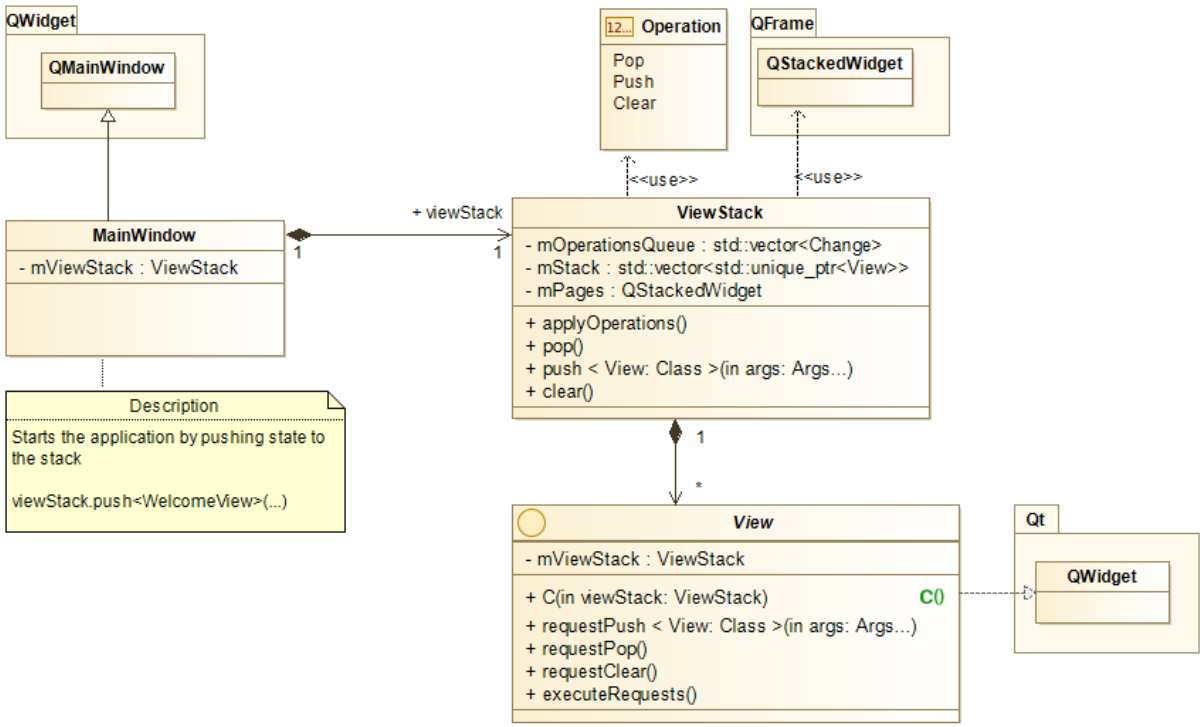


Figure 2: Simplified class diagram of Nizer application

4.3 Main window and ViewStack

The main window uses the *ViewStack* class to manage views – or, in other words, to operate on them freely by popping them of the stack or pushing them onto the stack.



4.4 Views

Views here are responsible for each individual transition of the application to a different part of it. Although they are displayed in the same window, they represent completely different parts of the application. Such views may want to pass data to each other, stay on the stack to avoid losing data, or pop of the stack when no longer needed.

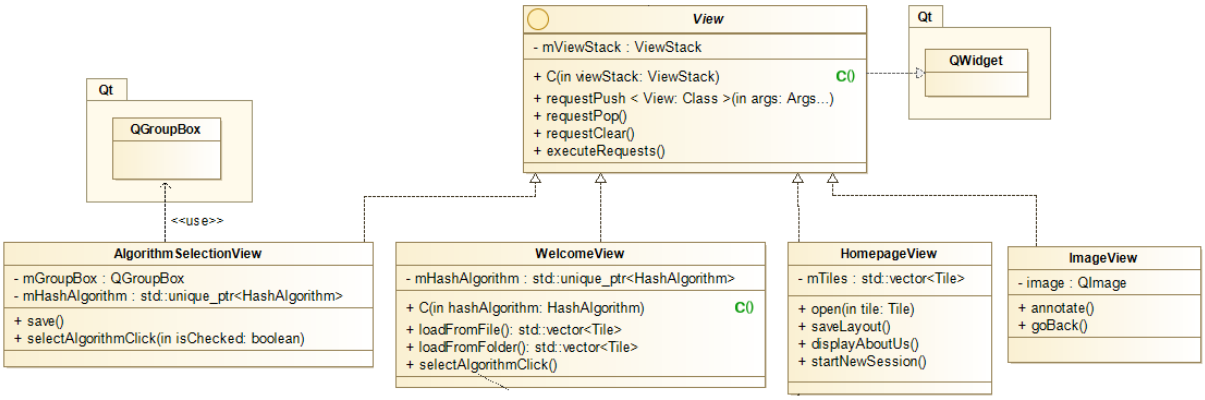


Figure 3: Views inside the Nizer Application

4.4.1 AlgorithmSelectionView

A view that allows the user to select one of the available algorithms from the *HashAlgorithm* interface to then pass it to the subsequent views.

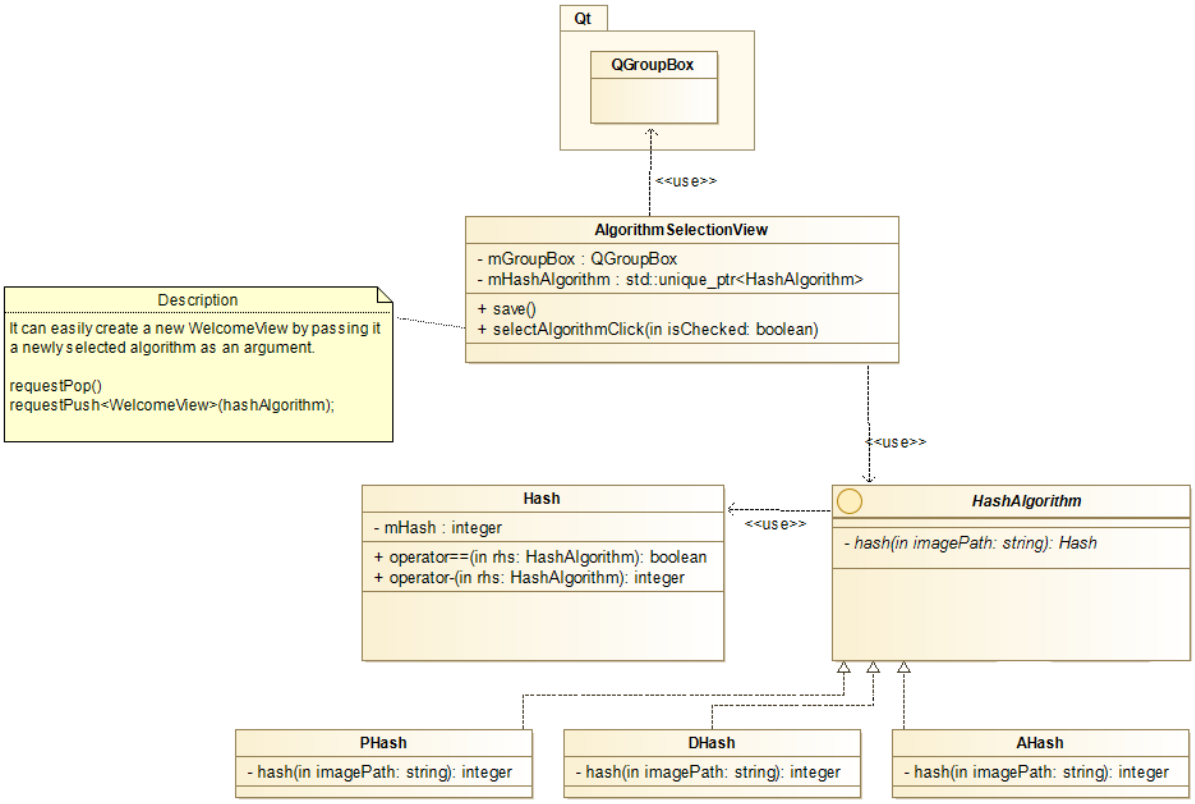


Figure 4: Structural model of the Algorithm Selection View

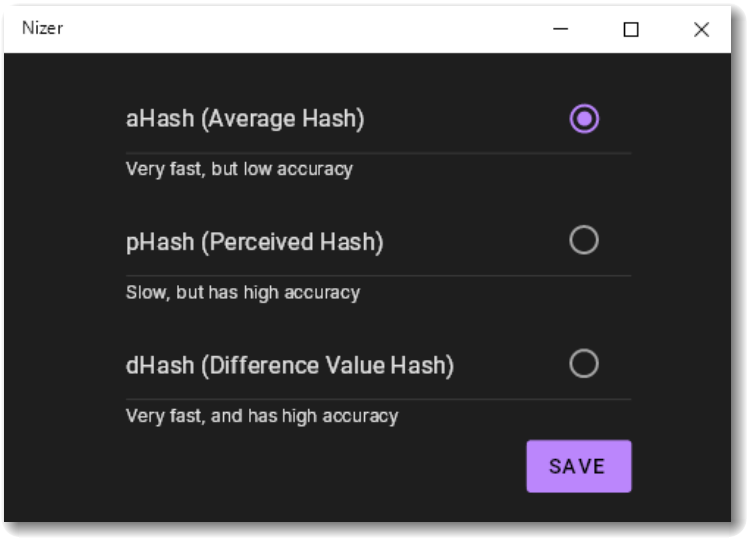


Figure 5: Graphical prototype of the Algorithm Selection View

4.4.2 WelcomeView

A view that allows to read images by pointing to the appropriate folder on disk, or by reading the save from an other session. It also allows to switch to the view to one related to grouping.

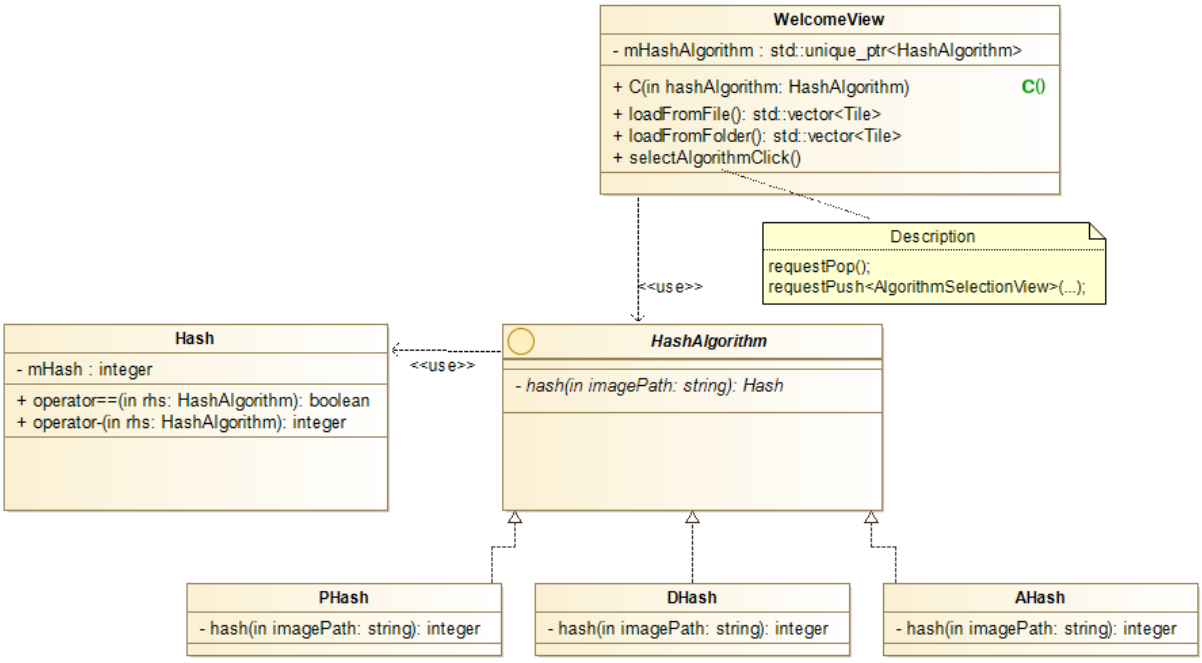


Figure 6: Structural model of the Welcome View

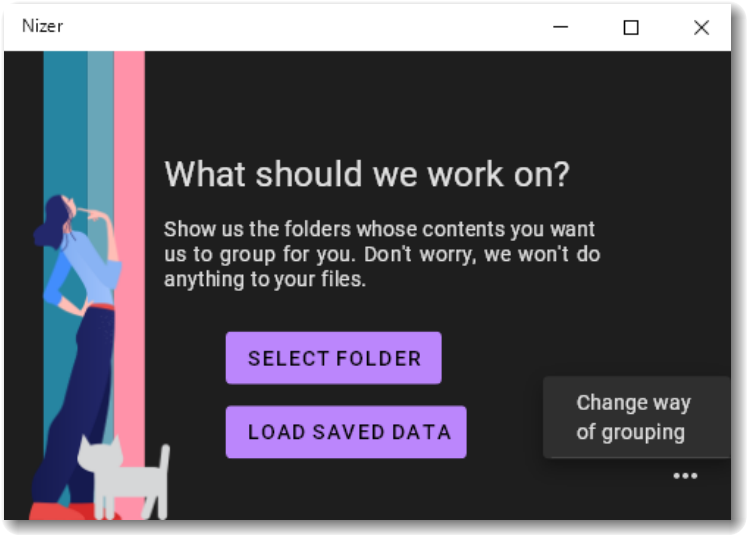


Figure 7: Graphical prototype of the Welcome View

4.4.3 ImageView

A view that displays an image selected by the user. Additionally, it allows to caption it with user's annotation, which displays when they re-open the image. Annotations are also saved when the session is saved to a file.

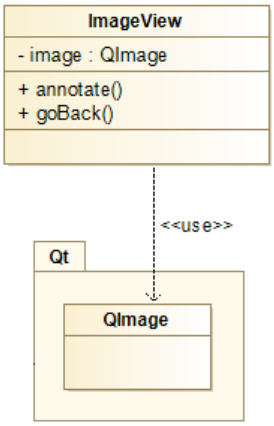


Figure 8: Structural model of the Image View

This page intentionally left blank

4.4.4 HomepageView

A view that gives the user an ability to preview folders with grouped data, i.e. clusters. In the slide-out side panel of the window, the user has several options to choose from, such as starting a new session, saving the session, or displaying information about authors.

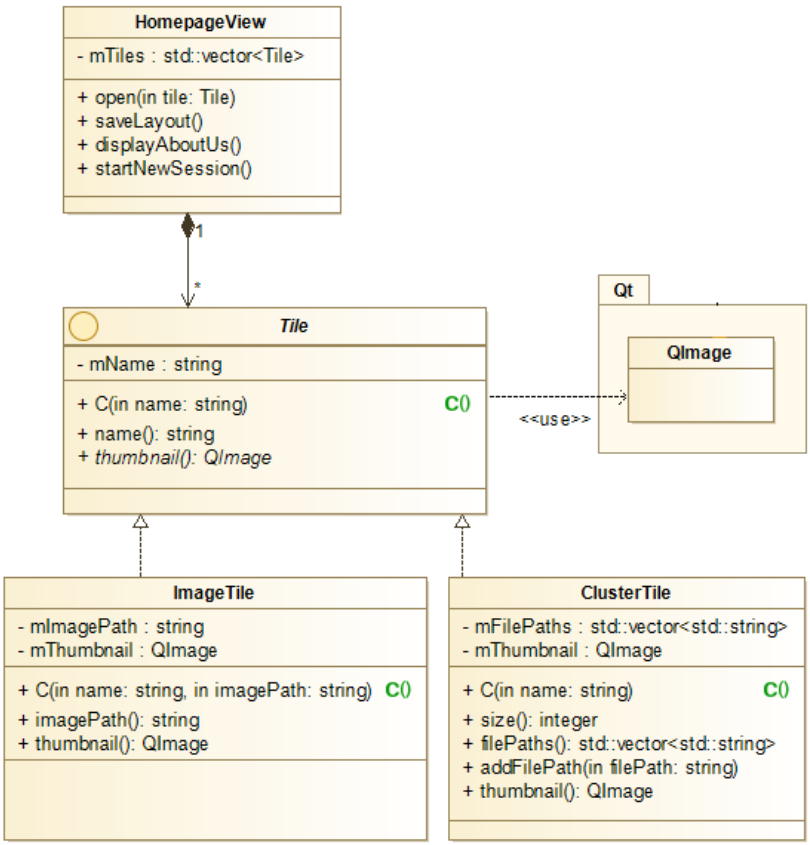


Figure 9: Structural model of the Homepage View

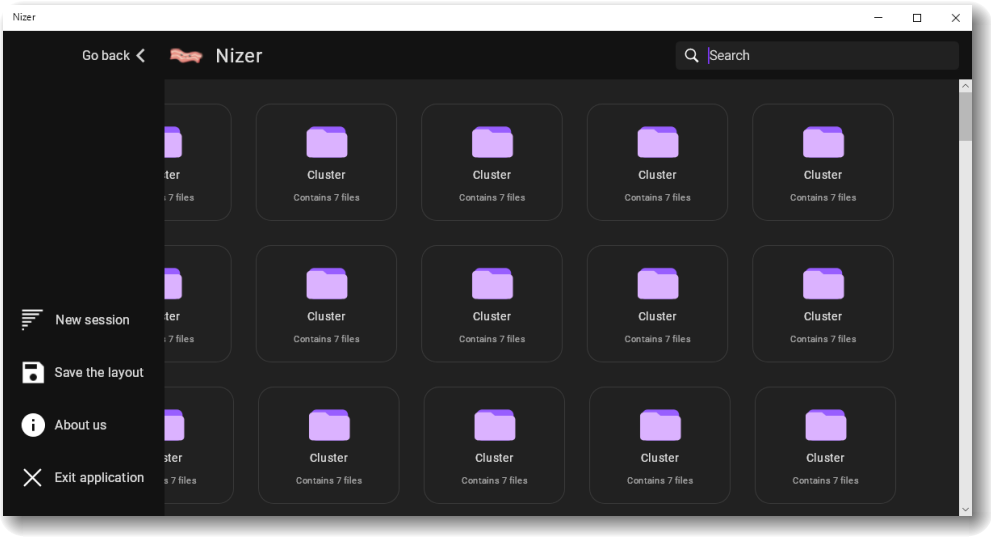


Figure 10: Graphical prototype of the Homepage View

4.5 Full class diagram

The full class diagram is heavily focused on implementations, including details such as technologies used or even frameworks.

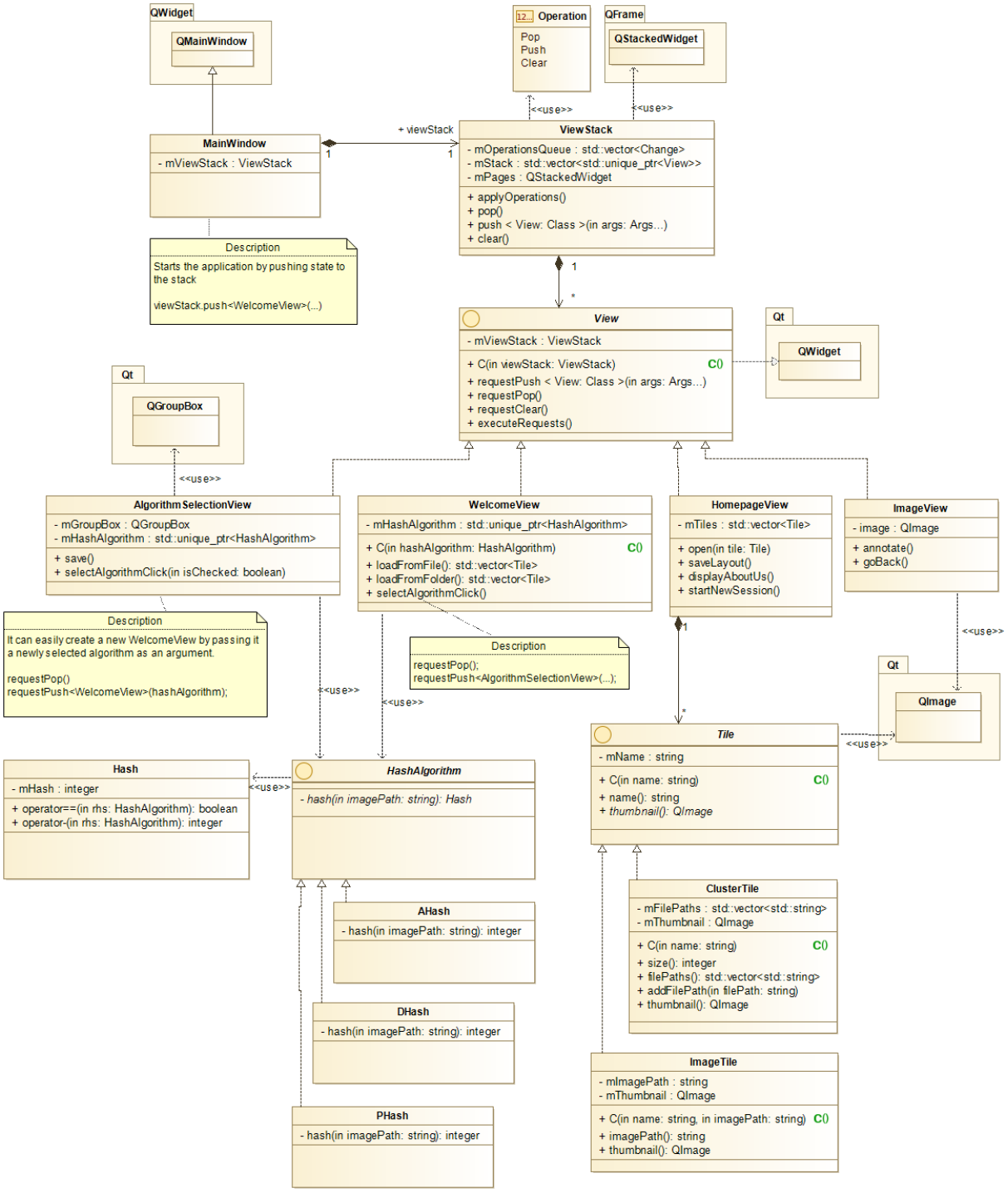


Figure 11: Class diagram of Nizer application

5 Behavioural modelling

To give a broader view of how the application works, please study Subsection *5.1. Activity Diagram* to understand the broader context as well as the previous diagrams in Section *4. Structure modelling* and *3. Use case modelling*. The activity diagram is here as an help to better understand what the sequence diagrams will represent.

5.1 Activity Diagram

The diagram attempts to show the flow of the entire application and serves as an extension of what can be seen in Section *3. Use case modelling*.

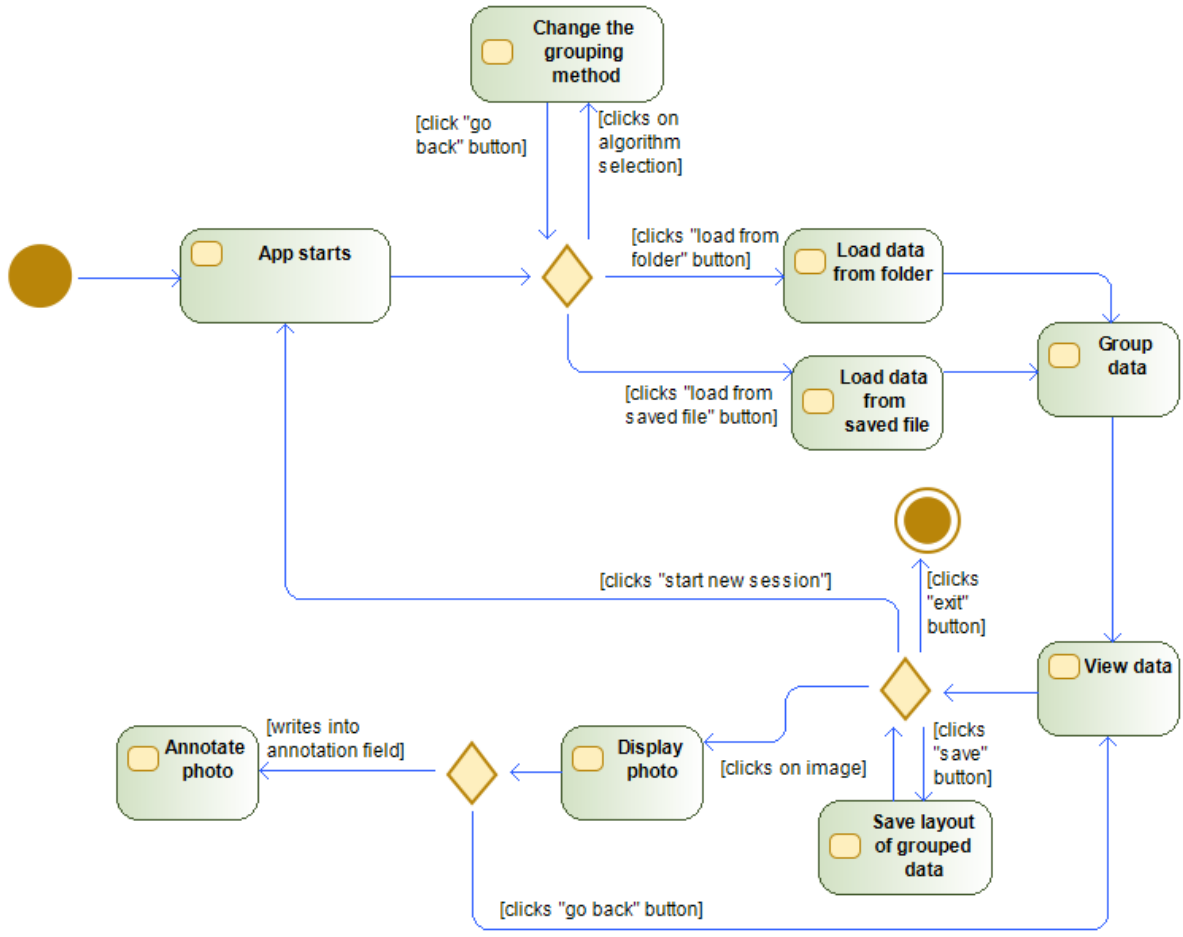


Figure 12: Activity diagram showing the full flow of the Nizer application

5.2 Sequential Diagrams

Remember: Sequence diagrams assume prior familiarity with use case diagrams, class diagrams and activity diagrams.

5.2.1 Startup of the application

The start of the application begins with the user launching the application. First, the main window is created, which then creates a `ViewStack` and pushes the start view (which is `WelcomeView`) onto its stack by adding a request to push the `WelcomeView` state to a `FIFO` queue. The changes are not yet visible until the `applyOperations()` method is called, after which the stack executes the requests put on the stack. In this case it creates a `WelcomeView`, which is displayed in the `MainWindow` because it is on top of the stack.

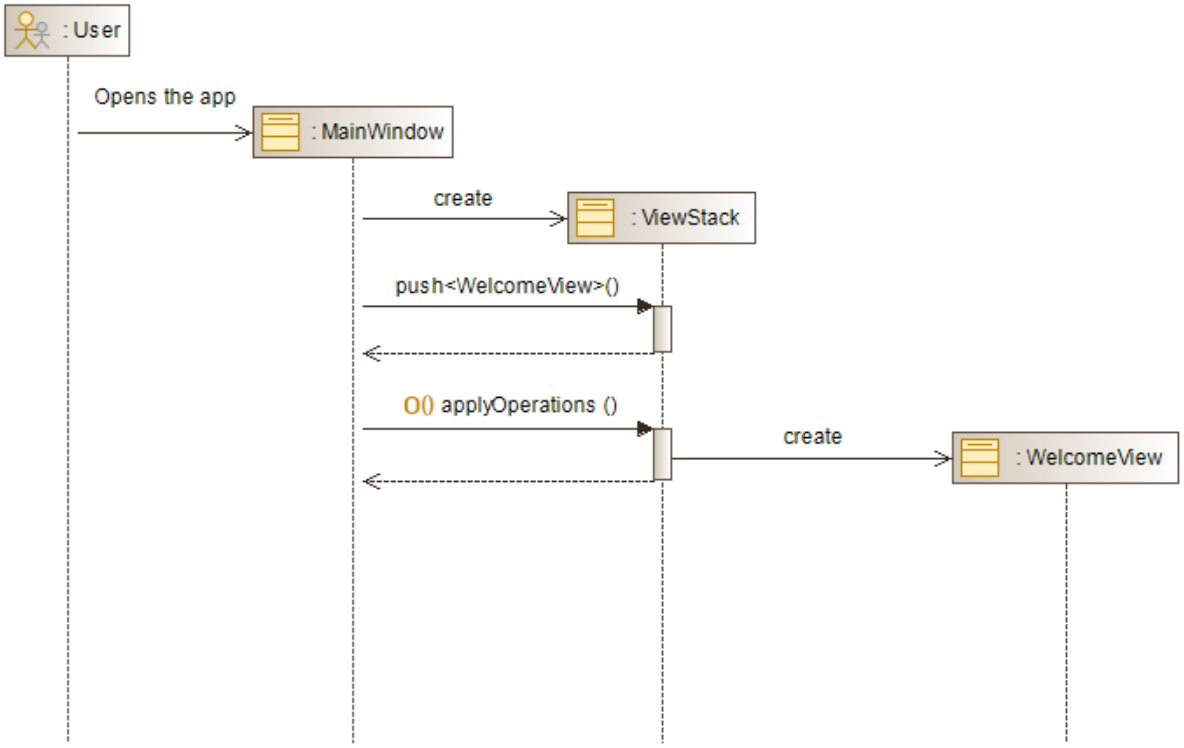


Figure 13: Diagram showing what happens (its initial state) when user launch the Nizer application

This page intentionally left blank

5.2.2 Data grouping

Data grouping occurs when the user indicates to the program the folder on which data clustering operations should be performed. Implicitly then, the user communicates with the view, which sets the operations in a `FIFO` queue:

1. The view pops itself from the stack – this step would be impossible without the `FIFO` queue, because if the operation were performed immediately the view would remove itself at this point and could not push the view onto the stack.
2. Push a new view onto the stack, along with passing it all the data about the clusters it created. `HomepageView` only displays this data.

Then the `executeRequests()` method calls the `applyOperations()` method to the `ViewStack`, which performs the operations from the queue and destroys the `WelcomeView`, as the `ViewStack` is its only owner. Then it creates a new view called `HomepageView` which, when created, is on top of the stack and is displayed in the main window.

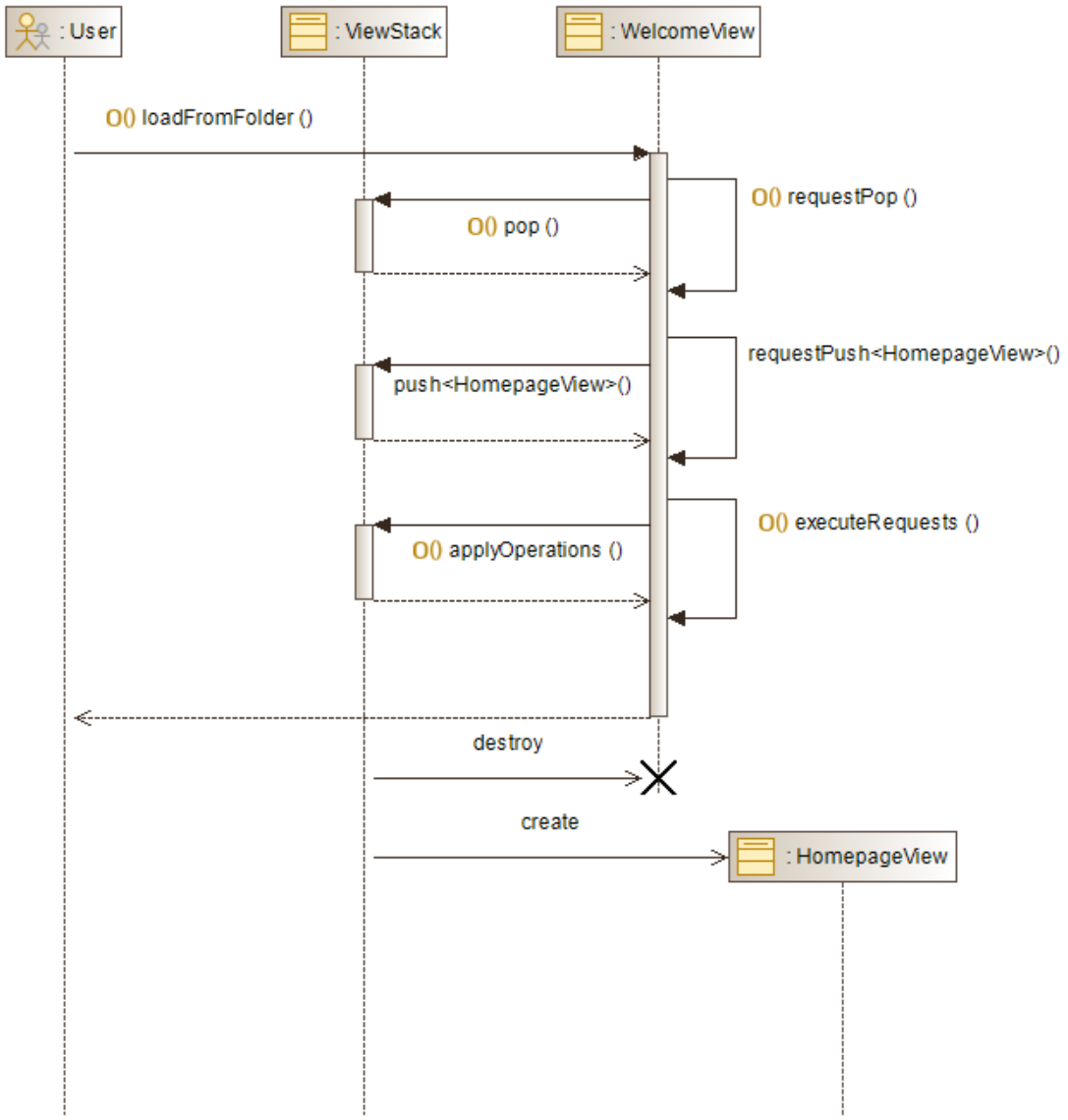


Figure 14: Diagram showing the state of data grouping and moving to the view of displaying clusters

5.2.3 Image Display

The image display benefits from the ViewStack action the most of all other cases. It allows to easily change the view and then return to the previous view in the identical state it was in before.

1. When the user opens one of the images, the `open()` method is called. This method adds a push request for an `ImageView` to the FIFO queue and calls `executeRequests()`. At no stage does it pop the current state off the stack, so it remains on it.
2. `ViewStack` creates an object of class `ImageView`, which from now on is on top of the stack and will be visible in the main window.
3. When the user wants to return, the user calls the `goBack()` method, which does nothing more than pop the `ImageView` off the stack. This will cause the `ImageView` to be removed and the `HomepageView` to take its place as the HomepageView is the view that was under the `ImageView` on the stack. Because the `HomepageView` is again on top of the stack it will be displayed in the main window. Thanks to the fact that the `HomepageView` has not been removed at any stage, it will resume its state as it was before the `ImageView` was created.

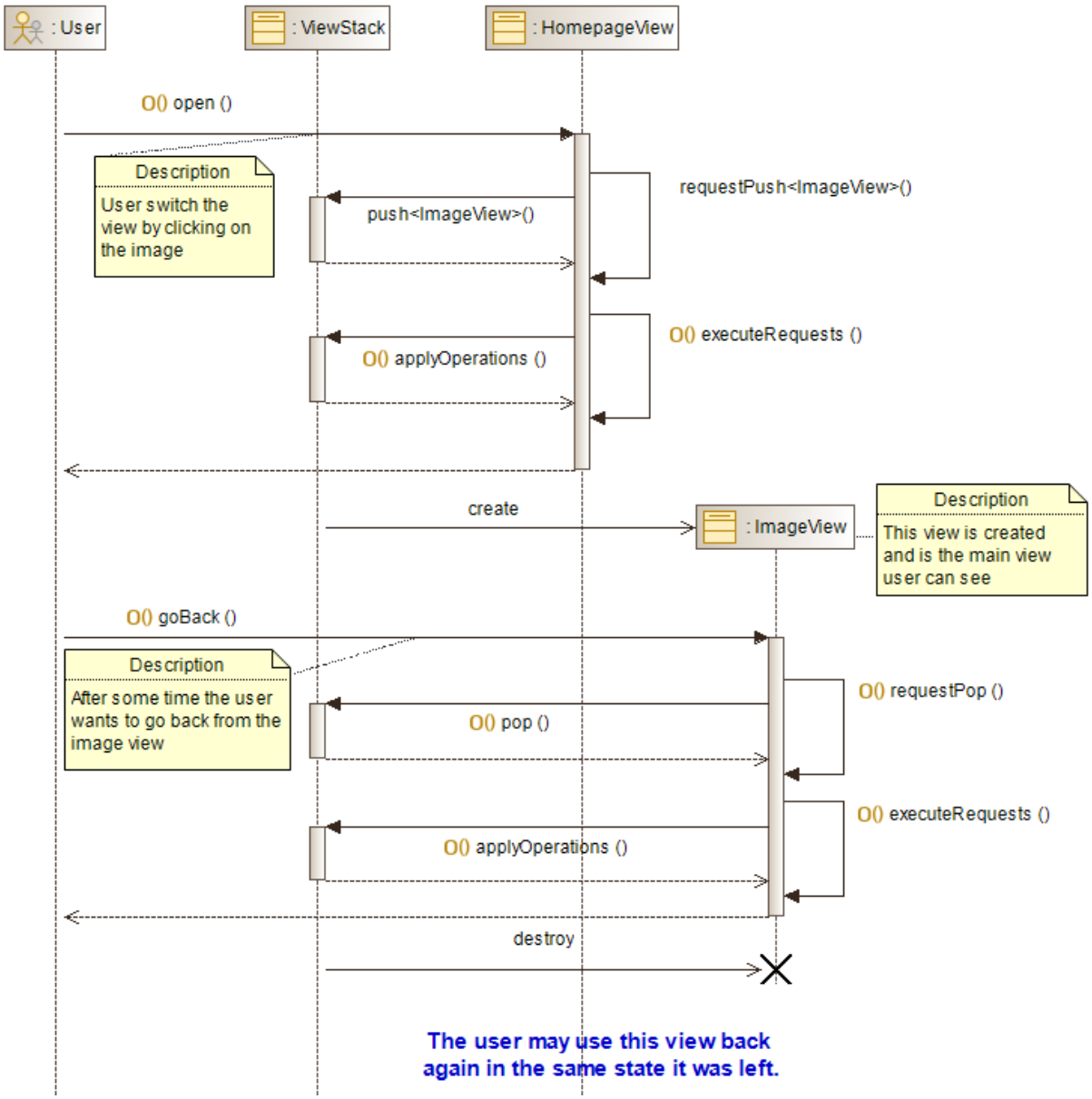


Figure 15: Diagram showing the process of displaying an image and returning to the previous state

5.2.4 Selecting new algorithm

Choosing a new algorithm is essentially a simple thing. In case the user wants to change the algorithm then the user calls the `selectAlgorithmClick()` method in the `WelcomeView`. It is designed to move the user to the algorithm selection view, so it:

1. Adds a request to the FIFO queue to drop itself from the stack.
2. Then it adds request to push the `AlgorithmSelectionView`.
3. Finally, it executes all the requests, by which `ViewStack` in turn removes the `WelcomeView` (by popping it from the stack) and adds the `AlgorithmSelectionView` to the stack, where the user can select the algorithm and then pass it back to the `HomepageView` (by popping itself from the stack and pushing in a new view).

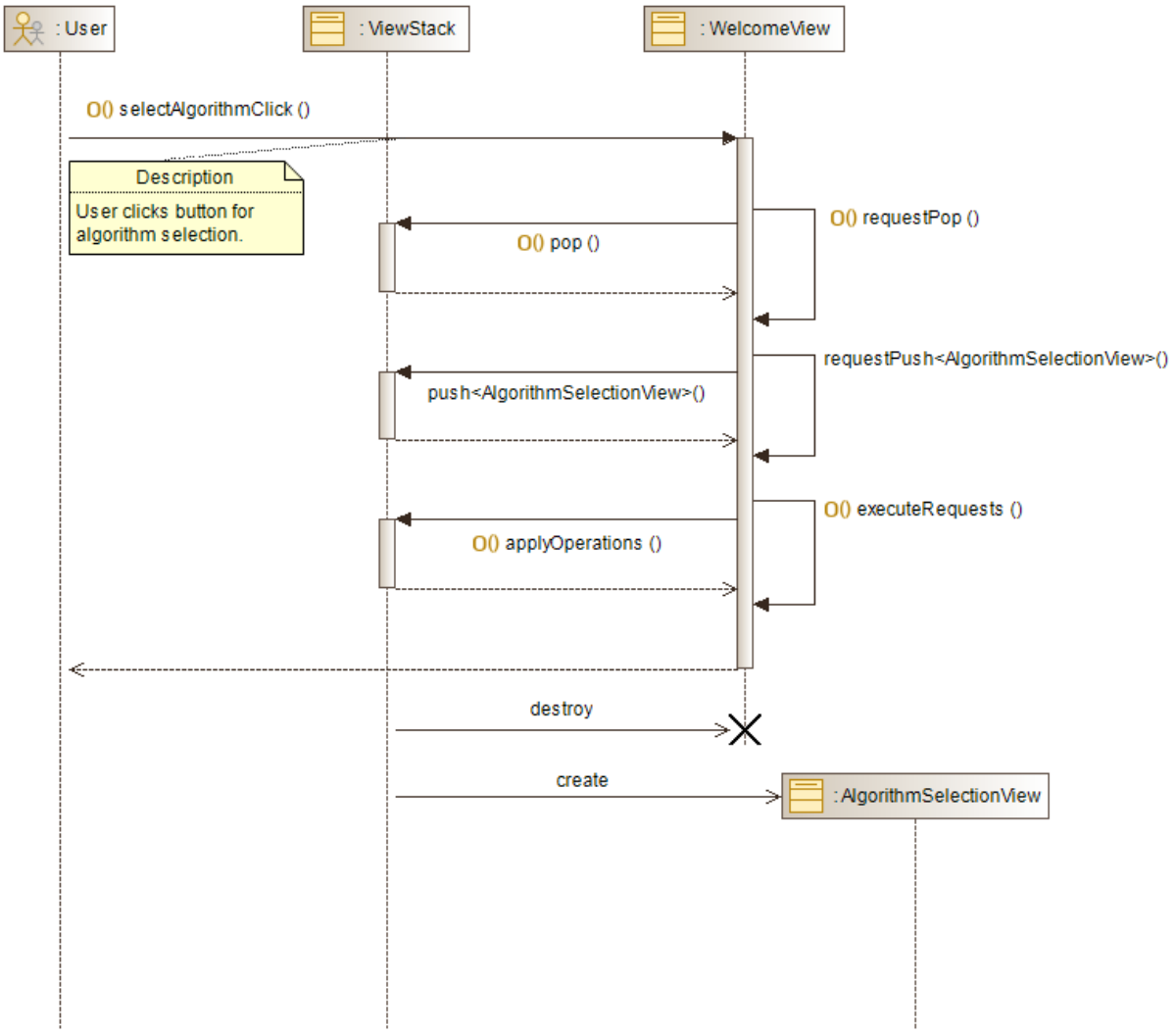


Figure 16: Diagram showing the process of selecting a new algorithm

6 Test Planning

This test plan defines the general ways of testing the application starting from the tools and frameworks used, to the ways of testing, the types of tests and even the list of tests needed to be performed. It is explained why the application is being tested in this way, and the risks are given.

6.1 Scope of testing

The application is used more by an individual, so the testing scheme should provide a reasonably pleasant, clear and responsive application for the user. The priority is the overall impression of interacting with professional software and providing at least the basic promised functions.

6.1.1 In scope

Key features that should be tested are:

1. Responsive and functional GUI, which should not cause any problems for the user on any platform used
2. The application should execute its basic objectives in a correct way and be open to change, while remaining resilient to uncontrollable and unexpected changes

As the application is rather intended for an individual user, it should take care of its user comfort and functioning interface, which if poorly implemented can most easily discourage a standard and often untrained IT user. For this purpose, **UI Tests** are necessary to ensure that the interface allows to navigate the application correctly and does not block its functionality. The functionalities mentioned here must work correctly and be adapted to changes related to user feedback. At the same time, we should be sure that functionalities are as intended so as not to confuse and frustrate the user. This will be the responsibility of **Unit Tests and Module Tests**.

Test levels

In summary, the priorities here are:

1. Unit Tests
2. Module Tests
3. UI Tests

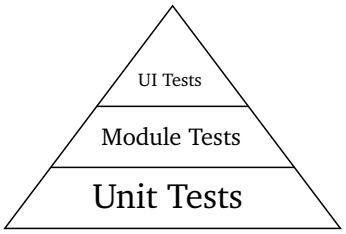


Figure 17: Test Level Priorities

6.1.2 Out of scope

Individual use does not imply parsing huge amounts of data impossible to collect by an ordinary user. There is no need for Load/Stress Tests, as performing them would be too expensive with respect to what is ultimately needed by the end user.

This page intentionally left blank

6.2 Test Procedure

To ensure good product quality, testing will be done on two levels: manual testing and automated testing. All tests are functional tests.

6.2.1 Automated tests

Automated testing will be done through two frameworks:

- GTest and GMock, which will allow to write unit tests and module tests. In this way, they will ensure that the implementation conforms to the specification, and any new changes do not introduce unexpected changes in behavior (known as Regression Testing).
- Qt Test will allow for very basic automated interface testing. It will ensure that important interface elements such as buttons work properly and at no stage will the user be stuck on the screen with no output or missing functionality.

Test Objective	**Ensure proper target-of-test functionality**
Technique	Each use case should be executed, its flow and functioning using both valid and invalid data. • For valid data, the program result should be predictable, consistent, and consistent with the documentation. • In case of erroneous data, the application should correctly report the errors with appropriate messages and warnings.
Entry Criteria	• The test environment should be properly prepared and be operational.
Completion Criteria	• All planned tests should be performed • All automated tests should pass and work.

Table 2: The way to perform automated unit, module and interface tests

This page intentionally left blank

6.2.2 Manual tests

Manual testing will cover what automated testing won't – this is mainly about system testing, which is done from the end user's point of view.

Test Objective	**Ensure proper target-of-test functionality**
Technique	Every use case should be carried out, both proper interface layout and responsive design should be checked. • The application should display interface elements as originally intended – correct positioning and responsiveness of elements (e.g. correct behavior with respect to window scaling). • Elements should always be accessible to the user (e.g. scaling should not make some functionality no longer available). • For valid data, the program result should be predictable, consistent, and consistent with the documentation. • In case of erroneous data, the application should correctly report the errors with appropriate messages and warnings.
Entry Criteria	• All automated tests should pass correctly. • The test environment should be properly prepared and be operational.
Completion Criteria	• All planned tests should be performed • All high priority bugs should be fixed. • Test results should be discussed and approved.

Table 3: The way to perform manual system and interface tests

This page intentionally left blank

6.2.3 Items to be tested

The number of test cases should be increased by the programmer responsible for testing.

Item to test	Test description
Loading data from folder	The user should be able to load data (create clusters) based on images from the specified folder • Loading data from a folder should create clusters whose sum of images should equal the sum of images in the specified folder. • Among the clusters, each image should occur only once in total. • Unsupported formats should not be present inside the clusters
Algorithm selection	The user should be able to change the algorithm, which should directly affect the algorithm used to create clusters when loading data from the folder
Saving	The user should be able to save the state of the application in order to load it later. The state after loading should be the same. This applies to the annotation state, cluster order, number of images, and even the arrangement of images.
Loading	The user should be able to load previously saved application state. If some files are missing, it should notify the user about it. Application state should be preserved. This applies to the annotation state, cluster order, number of images, and even the arrangement of images. Loading should bring the HomepageView to the screen.
Annotations	The user should be able to add the annotation to the image and display it again later in the same state. Even after restarting the application and loading a save.
New session	The user should be able to start a new application session. The new session should display WelcomeView again.
Exit button	The user should be able to terminate the application via the UI button
Image click	Clicking on the image should display the correct data and image to the same one shown in the thumbnail. An ImageView should appear on the screen. The back button should return to the view with images inside cluster.
Clicking on a cluster	The cluster should correctly display all of its images – both the correct thumbnails and the names, extensions, and number of images. The back button should return to the clusters view.
Side Menu	It should be possible to expand the menu and all buttons should be working, be interactive and accessible to the user.
Interface responsiveness	The interface should scale correctly. Buttons should be accessible to the user, and text should be readable.

Table 4: Various scenarios to be tested

6.3 Unit Tests

The unit tests are divided as tests for each class of the application.

6.3.1 ViewStack

Test ID	Test Case	Test Description	Expected result
VS01	Correct change of current state	When the operation of pushing a new state on the stack is called, the current application state should be changed.	The current application state has changed to the provided view.
VS02	Popping state from the stack when it is empty	When the stack is empty, the pop operation should be forbidden and should result with an error message.	An exception should be thrown.

Table 5: Unit test descriptions for ViewStack classes

6.3.2 AlgorithmSelectionView

Test ID	Test Case	Test Description	Expected result
AS01	Properly saved algorithm	The selected radio button should match the hash algorithm passed to the WelcomeView constructor. The test should be parameterized.	For a map with a function for a given button selection to the corresponding algorithm, the individual executions should match.

Table 6: Unit test descriptions for AlgorithmSelectionView classes

6.3.3 WelcomeView

Test ID	Test Case	Test Description	Expected result
WV01	Choice of algorithm	The button responsible for the ability to select an algorithm should want to call `requestPop()` and `requestPush` with `AlgorithmSelectionView`	Both methods correctly call the ViewStack
WV02	Loading from a folder	Loading from a folder should pass valid files to the hash function for the input data. A transition between the view and the `HomepageView` is also called.	Just check that the expected data is passed to the hash function and that `requestPop` and `requestPush` from `HomepageView` are called.
WV03	Loading from file	The method should correctly interpret the save file with data such as files or set annotations. It should then call `requestPop()` and `requestPush()` with the `HomepageView`.	The individual methods should be called on the `ViewStack`, and the correct values should be passed to the `HomepageView`.

Table 7: Unit test descriptions for WelcomeView classes

6.3.4 HomepageView

Test ID	Test Case	Test Description	Expected result
HV01	Starting a new session	The *ViewStack* class should be mocked. When the new session is started the *requestPop()* and *requestPush()* functions should be called.	When both methods were called, test is considered as successful.
HV02	Saving the layout	When the layout is being saved, the proper file with all image paths and its corresponding annotations should be created. All image paths and annotations in the created file should be the same as the one which are stored in the *HompepageView*.	The content of the file is identical to what *HompepageView* stores

Table 8: Unit test descriptions for HompageView classes

6.3.5 ImageView

Test ID	Test Case	Test Description	Expected result
IV01	Successfully adding annotation,	After the annotation is added, it should be the same as the expected annotation.	Both annotations are equal.
IV02	The return	The return function should call `requestPop`.	Check if `requestPop` is called once after `goBack()` is called.

Table 9: Unit test descriptions for ImageView classes

6.3.6 ImageTile

Test ID	Test Case	Test Description	Expected result
IT01	Check getter methods	Construct and initialize an object with expected values of image path and thumbnail. The getter methods should return the expected values.	Value returned from getter function equals expected value.

Table 10: Unit test descriptions for ImageTile classes

6.3.7 ClusterTile

Test ID	Test Case	Test Description	Expected result
CT01	Addition of invalid file paths	Adding a path to a non-existent image should end up with an exception being thrown.	Exception is raised.
CT02	Addition of valid file paths	Add multiple instances of valid paths to the created cluster instance. The returned paths from the cluster should be compared with the expected paths	The expected paths are equal to the returned paths
CT03	Check getter methods	Construct and initialize an class object with expected values of image path and thumbnail. The getter methods should return the expected values.	Value returned from getter function equals expected value.

Table 11: Unit test descriptions for ClusterTile classes

Test ID	Test Case	Test Description	Expected result
HA01	Hash comparison succeed	Two identical hashes when compared should return true	Hashes are the same
HA02	Hash comparison failed	Two different hashes when compared should return false	Hashes are different
HA03	Correct hash difference	The difference between the two hashes should be the same as the expected difference	Expected hash difference equals calculated one
HA04	Invalid hash difference	The difference between two hashes should not be calculated when incorrect hash values are provided	An exception is raised
HA05	Successful hash calculation	Computed hash is the same as the expected hash value	Expected hash value equals calculated one
HA06	Hash computation failure	When an incorrect path to an image is provided, the hash should not be calculated.	An invalid image path exception is raised. The hash value is not computed

Table 12: Unit test descriptions for HashAlgorithm classes

This page intentionally left blank

6.4 Module Tests

Test ID	Test Case	Test Description	Expected result
MT01	Loading data from folder	A number of images loaded from the folder in the *WelcomeView* should be equal to the sum of all images in the clusters.	Number of loaded images equals images in the clusters
MT02	Selection of the new algorithm	When an algorithm is selected in *WelcomeView*, it should be saved and its selection should be reflected in the appearance of the created clusters in *HompageView*.	Clusters for specific algorithms and cases should be different
MT03	Annotation preservation	Saving the annotation in the `ImageView` should be preserved when user return to the `HomepageView` and reopen the same image in the `ImageView`.	The annotation in both cases should be the same

Table 13: Description of the Module Tests

This page intentionally left blank

6.5 UI Tests

6.5.1 AlgorithmSelectionView

Test ID	Test Case	Test Description	Expected result
UI_AV01	Correctly returned algorithm	Selecting the appropriate radio button by clicking should affect the algorithm passed to `WelcomeView`.	Buttons correctly influence the passed algorithm.
UI_AV02	Responsive interface	Buttons and radio buttons should be clickable. Buttons should respond to the cursor – through animation or displaying a hand with a finger.	As in the test description
UI_AV03	The save button should respond	Clicking the save button should change the view to another.	Changing to a different view.

Table 14: UI test descriptions for AlgorithmSelectionView

6.5.2 WelcomeView

Test ID	Test Case	Test Description	Expected result
UI_WV01	Algorithm selection	Clicking on algorithm selection should change the view to `AlgorithmSelectionView`.	The view should change.
UI_WV02	Loading buttons	Clicking the loading buttons should launch a window with a file/folder selection.	A file/folder selection window should pop up.
UI_WV03	Selecting a file/folder	Selecting a file/folder should result in the view changing to `HomepageView`.	Change the view to `HomepageView`.

Table 15: UI test descriptions for WelcomeView

6.5.3 HomepageView

Test ID	Test Case	Test Description	Expected result
UI_HP01	Successful application closure	When the exit button is clicked, the application should be terminated.	Application is closed
UI_HP02	Verify all buttons are clickable	All HomepageView button should always be clickable and responsive.	Buttons respond to clicks by enabling the appropriate action
UI_HP03	Correct scaling of the window	The window should not be a fixed size, the user should be able to scale the window according to their needs. When scaling, the buttons should stay in place.	As in the test description

Table 16: UI test descriptions for HompageView

6.5.4 ImageView

Test ID	Test Case	Test Description	Expected result
UI_IV01	Proper text wrapping	When the file path, or image name, is too long, the text should wrap. The text should be wrapped in such a way that it can be read at all times.	As in the test description
UI_IV02	Correct scaling of the image	The image should be scaled to the size of the frame holding it. When resizing the window, the frame holding the image should scale properly with the window.	As in the test description
UI_IV03	Adding annotation	When the add annotation button is clicked, it should be replaced with a text box.	As in the test description

Table 17: UI test descriptions for ImageView

This page intentionally left blank

6.6 Resources

A small project does not consist of many resources. Basic applications, not necessarily designed for large companies, are sufficient.

Name of process	Tool
Defect Tracking	Github Issues
Test Cases	**Qt Tests** for UI Tests **GMock, GTest** for Unit and Module Tests

Table 18: Table of tools used

Name of OS	Version of OS
Windows	7, 8, 10, 11
Linux	Latest Ubuntu, Latest Debian

Table 19: Table of operating systems on which the tests will be performed

6.7 Testing Process Risks

The project team is quite small, so the main risk is the temporary absence of any of the staff. The advantage of a small team is that they are well acquainted with the project, so the tests should be carried out correctly and efficiently.

Risk	Mitigation
A short deadline that will not allow for all testing.	Set test priority for each of the test activity.
Delays in fixing bugs	Daily meetings focused on helping to solve problems
Unexpectedly long bug fixing time, resulting in delays in delivery of new builds	More careful planning and time estimation

Table 20: Main risks of the project

Appendices

A Project Dictionary

A

aHash • Average hash algorithm.

C

Cluster • Cluster refers to a collection of images aggregated together because of a certain degree of similarities to other objects in their group.

D

dHash • Difference hash algorithm.

H

HDD • Hard Disk Drive is an electro-mechanical data storage device.

HashAlgorithm • One of the candidates for the class. It is an interface that implements the general behavior of a hash algorithm. This interface is useful since in an application the user has a choice between several hashing functions.

I

imageHash • An image hashing library. It implements several hashing algorithms.

N

Nizer • An application that allows to easily organize a huge number of photos based on their similarity.

P

pHash • Perception hash algorithm.

Perceptual Hash • It is a 64-bit number fingerprint of an jpg file, derived from various features from its content. A two hashes are "close" to each other when features of the images are similar.

Primary Actor • An actor that triggers a use case. In our case our primary actor is the user whose requests are served by the running application.

S

Secondary Actor • They are reacting to actions made by primary actor.

Session • Current or new instance of a running application.

SSD • Solid-State Drive it is a storage device typically using flash memory.

T

Tile • One of the candidates for a class. An interface that contains specific functions that an object must have to be a Tile. There will most likely be two implementing classes, ImageTile and ClusterTile.

U

USB • Universal Serial Bus. A popular portable data storage device.

Use Case • Describes actions that actors can take.

User • A user running the Nizer application. In our system is also defined as the primary actor.

V

ViewStack • One of the candidates for the class. This is a structure that controls what view the user currently sees in the application window. Thanks to the stack-like design it ensures the smooth transition between different views, such as from welcome view to the home page view.

View • One of the candidates for the class. Every view represents a separate part of the application. The application has several of them they are listed in the class diagram in the section *4.4*. Each sub-view must implement the generic View interface, which defines the basic functionality of each view.