Requirements and Analysis Document for OOPsie

Petter Blomkvist, Simon Engström, Simon Johnsson,
Axel Larsson, Love Lindqvist

October 24, 2021

Contents

1 Introduction

The project aims to create a personal realations manager where a user can store contacts and events to keep track of their relationships, both professional and personal. It should be possible to register contacts and their information, and categorize them with custom labels. It should also be possible to log events and choose what people the event concerns. There should also be a way to see all your events in a timeline/calendar view, to give the user a broader overview of what events are planned and have been logged.

The idea is that anyone who wants to log and keep track of relationships and personal events can use this application, which essentially could be anyone with the will to do so.

The application is especially useful for people that are forgetful or have very large social circles.

1.1 Definitions, acronyms, and abbreviations

- **PRM**: Personal Relations Manager.
- **User Stories**: A type of feature request written from the perspective of the application user.
- **GUI**: Graphical user interface. Allows users to interact with the software using a graphical interface.
- **User Wizard**: A tool for simplifying the selection and creation of a user.
- **JavaFX**: A GUI library for java.

2 Requirements

2.1 User Stories

2.1.1 Contacts US1 - Completed

As a person with an active social life, I want to be able to keep track of my contacts, and their basic contact information.

- Functional criteria
  - Can I view all my contacts?
  - Can I view the information of any specific contact?
- Non-functional criteria

- Can I add/change information of a contact?

2.1.2 Save/Load US11 - Completed

As a user I want my contacts and events to save so that I don't have to keep the app open at all times.

- Functional criteria
  - Can I save my current user data?
  - Can I load user data?
- Non-functional criteria
  - Will data be stored in the $HOME/.prm directory?

2.1.3 Events US3 - Completed

As a user I want to be able to plan and log social interactions, so that I can easily keep track of what has happened in my social life.

- Functional criteria
  - Can I add an event?
  - Can I edit an event?
  - Can I remove an incorrect event?
- Non-functional criteria
  - Can I view an event?

2.1.4 Tags US2 - Partially complete

As a user I want to be able to add tags to my contacts and my logged events.

- Functional criteria
  - Can I add a tag to a contact/event?
  - Can contacts have several tags?
  - Can an event only have one tag at a time?
  - Can I rename a tag?
  - Can I create new tags?
  - Can there only be one tag with a certain name?
- Non-functional criteria
  - Can I view what tags a contact has?
  - Can I view what tag an event has?

2.1.5 Notes US4 - Completed

As a user I want to keep notes of my relations so that I can review facts about them.

- Functional criteria
  - Can I submit a note?
  - Can I view my already submitted notes?
  - Can I edit an already submitted note?
  - Can I remove an already submitted note?
  - Can I see what time I added the note?
  - Can I sort my notes?
- Non-functional criteria
  - Will my notes remain unchanged if I don't change them myself?

2.1.6 Contacts in events US5 - Completed

As a user I want to be able to add contacts to events, in order to know who participated in my events.

- Functional criteria
  - Can I add a contact to an event?
  - Can I add several contacts to an event?
  - Can I edit the participants of an event?
- Non-functional criteria
  - Can I view what events a contact is associated with?
  - Can I view the contacts associated with an event?

2.1.7 Search US9 - Completed

As a user I want to be able to search through my submitted information so that I might find it more easily.

- Functional criteria
  - Can I input a searchword?
  - Does the search remove irrelevant results?
  - Does the search prioritize more relevant results?
- Non-functional criteria
  - Will I be able to view information without searching?

2.1.8 Statistics US6 - Completed

As a user I want to be able to see statstics over how I delegate events between my different contact types.

- Functional criteria
  - Can I extract event delegation statistics from the application model?
- Non-functional criteria
  - Can I view statistics of how many events I've had with my different contact categories?

2.1.9 Attachments US7 - Completed

As a user I want to save file attachments to contacts so that I can organize my files related to these contacts.

- Functional criteria
  - Can I add attachments to contacts?
  - Can I remove attachments from contacts?
  - Can I open the attachments in the default app for the given file type?
- Non-functional criteria
  - Can I view contacts' attachments at any time?

2.1.10 VCF parsing US8 - Completed

As a user I want to be able to import contacts from other sources.

- Functional criteria
  - Can I load in Contacts from external sources?
  - Do the contacts get added to my contacts?
  - Can I import multiple contacts at once?
  - Do the imported contacts include as much relevant information as possible?

2.1.11 Notifications US10 - Partially Completed

As a user I want to be notified about upcoming events in order to minimize the risk of forgetting.

- Functional criteria
  - Can I clear notifications?

  - Can I mute notifications?

- Non-functional criteria
  - Can I see a notification when it occurs?
  - Can I list all notifications?

2.1.12 Agenda US12 - Not implemented

As a user I want to se an agenda for the forseable future, to better plan ahead

- Functional criteria
  - Are all events loaded in properly?
  - Are events orderd according to time?
  - Is the timeline updated when the time of an event is changed?
- Non-fuctional criteria
  - Can I view all upcoming events?

2.1.13 Goals US13 - Not implemented

As a user, I want to be able to set up and track goals to move my life in the direction i want.

- Functional criteria
  - Can I set up a new goal?
  - Can I set an end date for the goal?
- Non-Functional criteria
  - Can I viwe my goals, and their due dates?

2.2 Definition of Done

Common criterion for all user stories is listed below.

- The code works on all three major operating systems, Windows, Mac, and Linux.
- All public methods are documented using JavaDoc, and classes contain a description.
- The code passes all tests, with 90%+ method/service and line coverage in model and services.
- It fulfulls the functionality described by the user story.

2.3 User interface

2.3.1 Whiteboard sketches

The first preliminary version of the GUI was sketched on a whiteboard. Figure 1 shows the view of all the created contacts in the CRM, while figure 2 shows the view for a single contact in the CRM. All elements of the GUI in these sketeches are subject to change since as the development proceeds, features will be added that might affect or extend the GUI shown in these sketches.

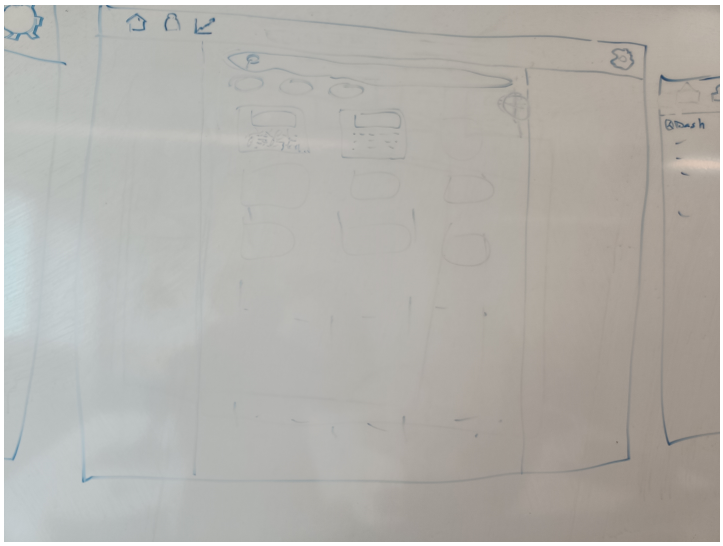

Figure 1: A scetch of the overview of all the contacts in the CRM, with the option to filter by tags.

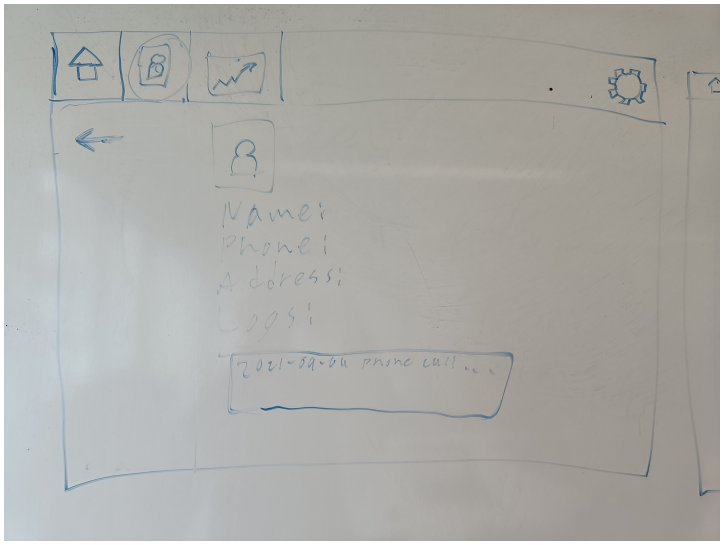

Figure 2: Preliminary view of a contact with basic information such as name.

Figure 3 shows a preliminary home screen. Figure (4) displays a statistics window, showing recent activity, statistics and more data.

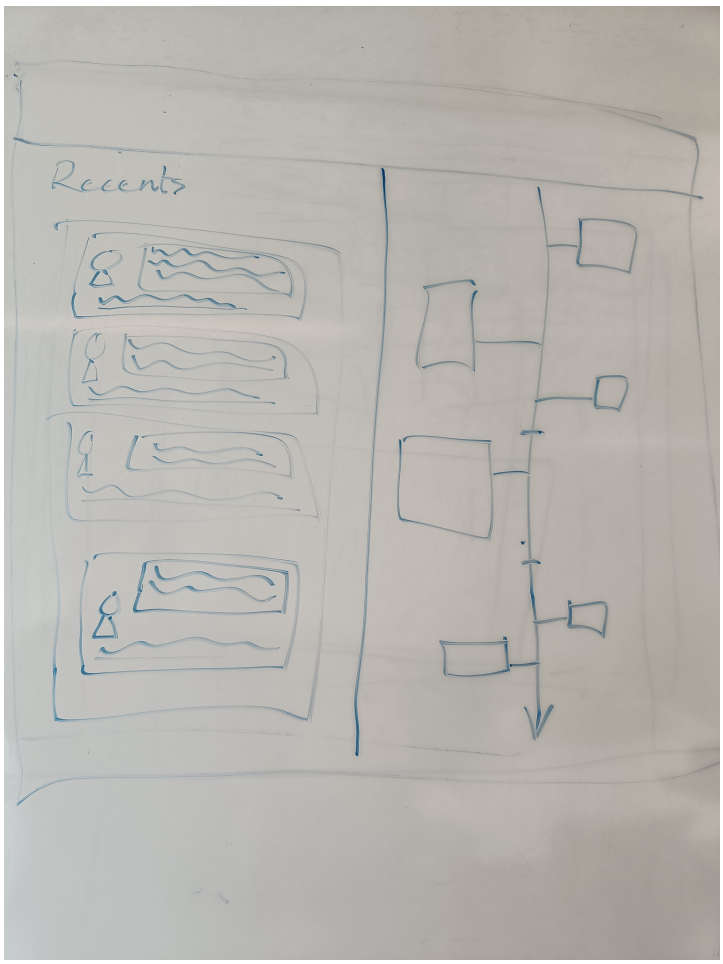

Figure 3: View from the home screen, showing recently viewed contacts and an overview of the timeline

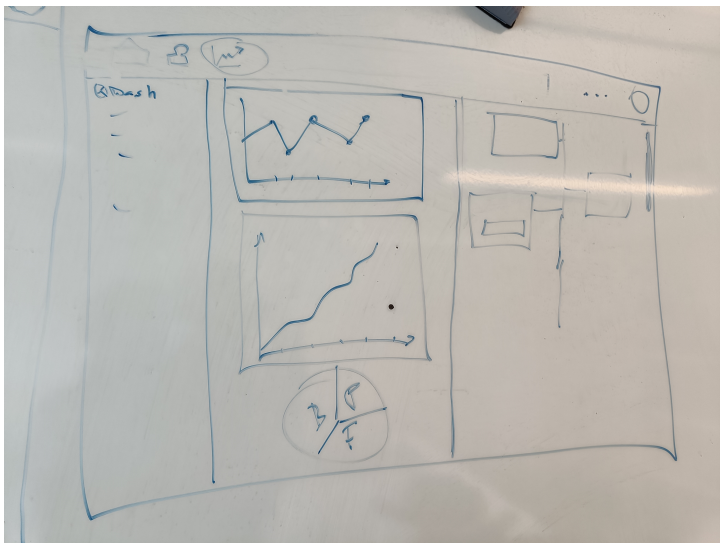

Figure 4: A sketch of the statistics board, showing activity and types of interations with contacts

In this version of the GUI, navigation would be performed by clicking on the icons of the navigation bar located at the top of the GUI. When viewing a contact in full-screen mode as in figure 2, navigation back to the contact-browsing view shown in figure 1 is performed by pressing the backwards-arrow in the top-left of the window, directly under the navigation bar.

2.3.2 Figma sketches

Based on the whiteboard sketches, a more detailed prototype of the GUI was made using the sketch-mode in Figma. The preliminary home screen now looks as follows in figure 5. This sketch shows a unlogged events tab instead of a recent contacts tab as sketched in 3.

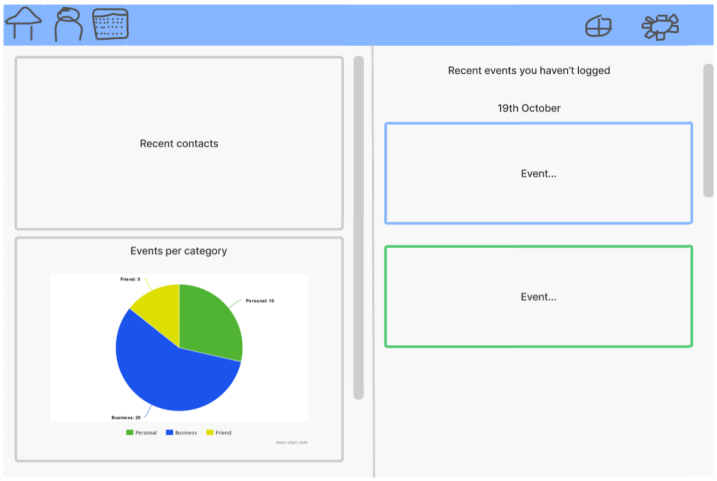

Figure 5: Preliminary home-page made in Figma.

The Figma sketch of the contact-browsing page as seen in figure 6 looks a lot like the one in figure 1. Featuring the same tag-based filtering features, a search bar and a grid of contact cards to browse. These cards include the profile-picture of the contact, the name and the category of the contact, color-coded, which makes for a simple and effective browsing environment.

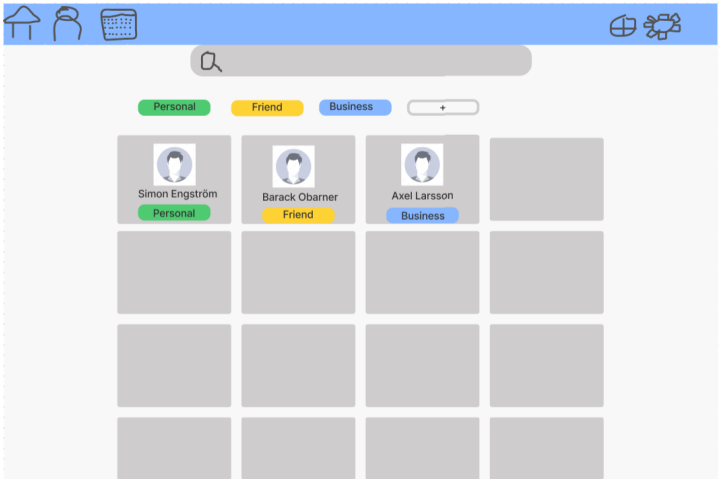

Figure 6: Preliminary contact-browsing page made in Figma.

The Figma version of the fullscreen contact-view is shown in figure 7. In contrast to the whiteboard sketch seen in figure 2 of this view, this sketch is more structured into sections and makes better use of the space to feature all of the information and functionality of the contact object.

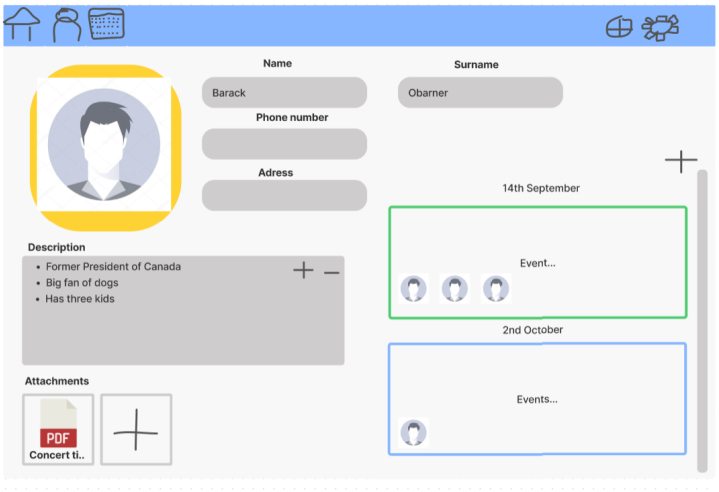

Figure 7: Preliminary fullscreen view of a contact made in Figma.

In the Figma sketch, we implemented the timeline in the form of a calendar to present the timeline in a more conventional and traditional manner, as shown in figure 8. Each column represents a day of the selected week, containing the events planned for that day. These event items contains, much like the contact cards in figure 6, basic information about the event, and the category of the event color coded into the border of the item, making it easy for the user to separate categories of events from each other.

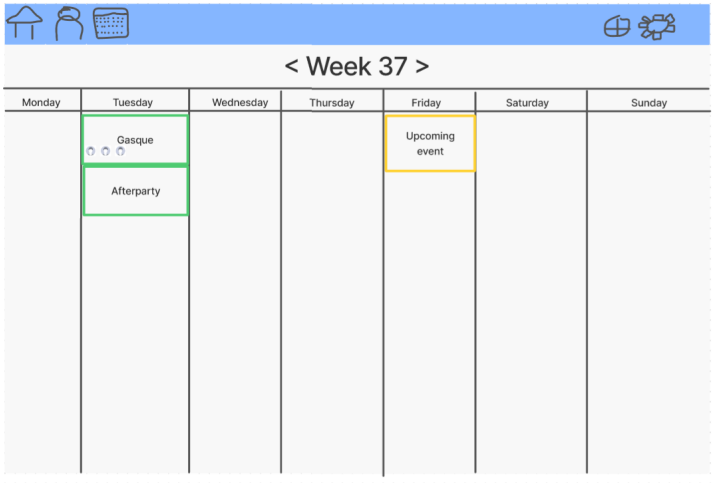

Figure 8: Preliminary sketch of the timeline-feature in the form of a calendar made in Figma.

In this version of the GUI, navigation through the interface is performed by clicking on the icons in the navigation bar located at the top of the interface. Events and contacts are added by clicking on the plus-sign in the top bar. To navigate back to the contact grid view rom the fullscreen contact view shown in figure 7, one has to press

the contact icon in the navigation bar.

There is no statistics page included in this sketch of the GUI, instead it is part of the home page shown in 5, although since it is only a preliminary sketch, this could be subject to change.

2.3.3 Final GUI

The final GUI was designed using JavaFX's default components with minimal styling changes.

The first view the user sees is the User Wizard, shown in Figure 9 which helps the application user create or select the correct user, since the application supports adding several users.

From this stage, the user can utilize the buttons in the top bar to navigate to the calendar page, the statistics page and the notifications page. This top bar is present in all parts of the GUI, but can be temporarily inaccessilble due to modal dialogs.

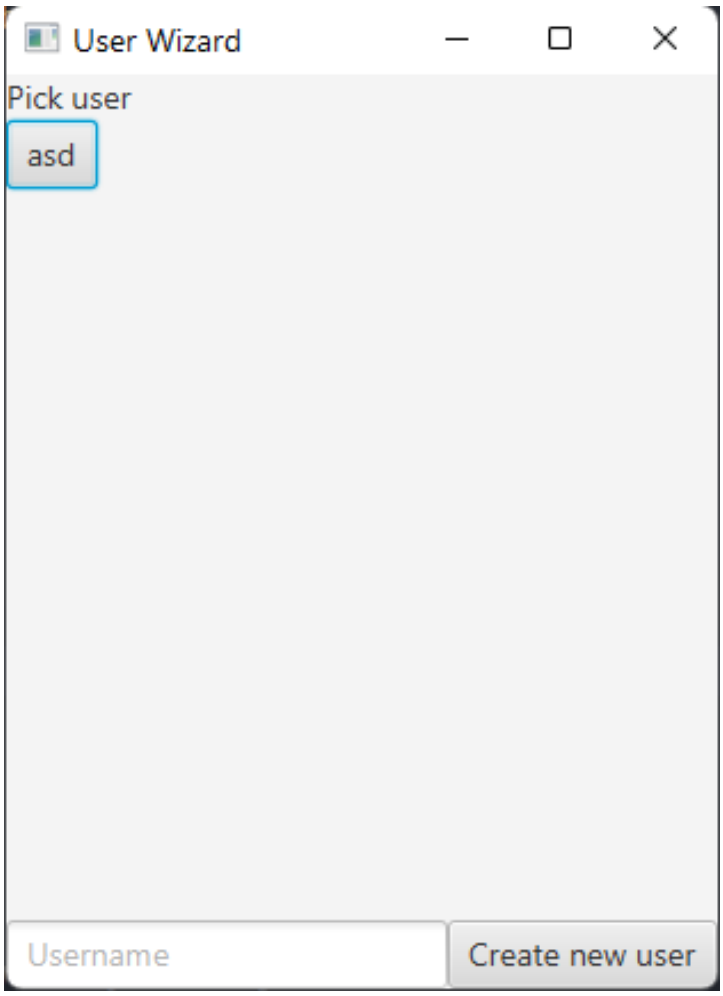

Figure 9: The GUI of the User Wizard

When a user is selected, the application user is presented with the home page, which is the contact page presented in figure 10.

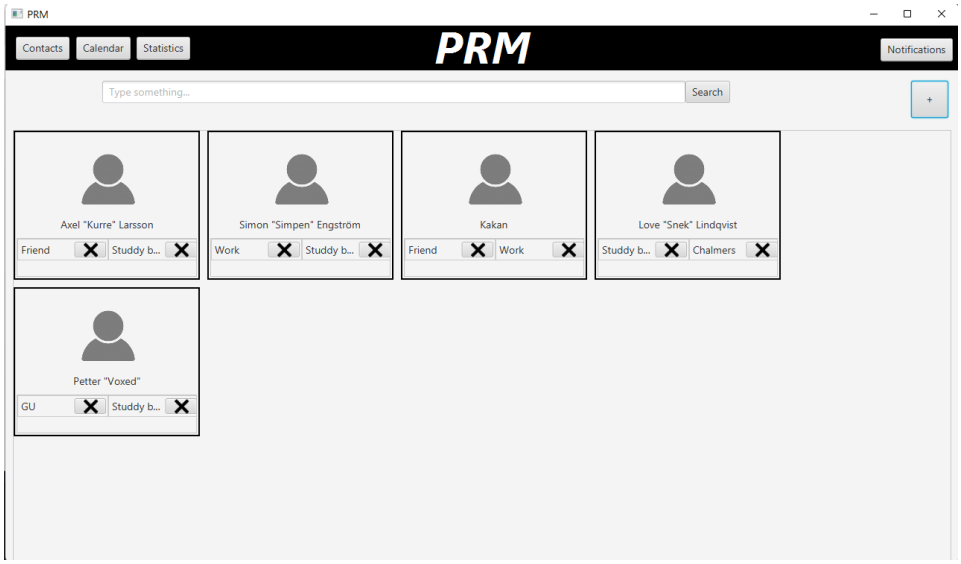

Figure 10: The final implementation of the contact overview page.

When clicking on a contact card, the user is presented with a view of all the contact's information, in an editable state, shown below in figure 11. To navigate out of this view, one can click on either the close-button in the top right, the done-button in the bottom right, or the gray area between the highlighted card and the edges of the window.

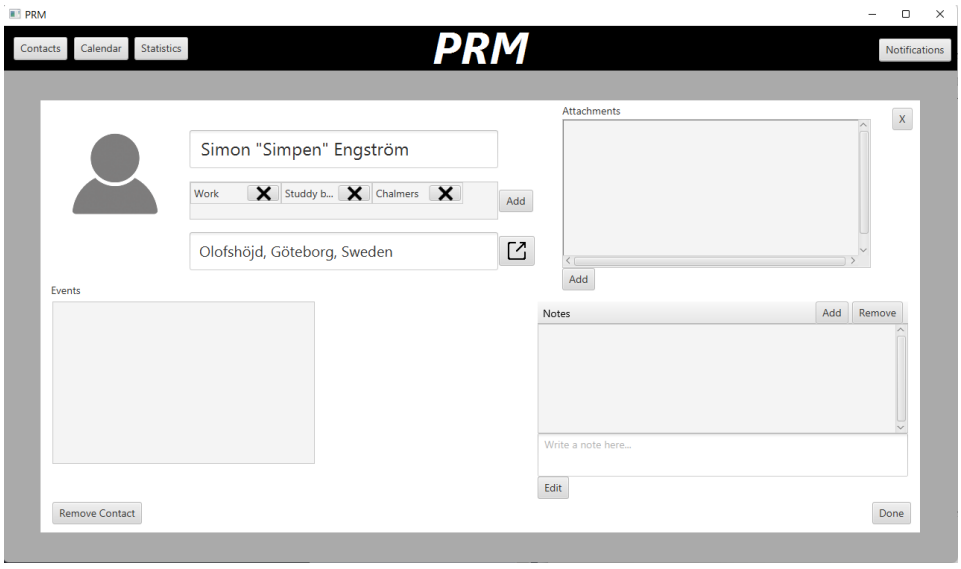

Figure 11: The final implementation of the contact view.

To create a contact, the user klicks on the plus button in the top right of the contact page displayed in Figure 10 and is presented with the dialog shown in Figure 12, where they can create a new contact by entering a name. The user can also choose to

import contact from a directory or file by clicking on either of the presented buttons, when doing so the user is asked to pick a file or directory to import from using the operating systems standard file picking dialog.

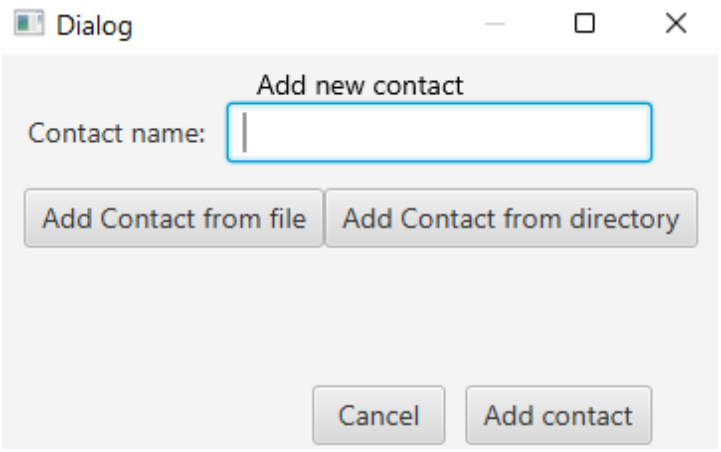

Figure 12: Final implementation of the dialog window used to create contacts.

When navigating to the calendar page seen in figure 13, the user is met with a traditional, week based calendar where events are sorted chronologically from top to bottom in each day's column.

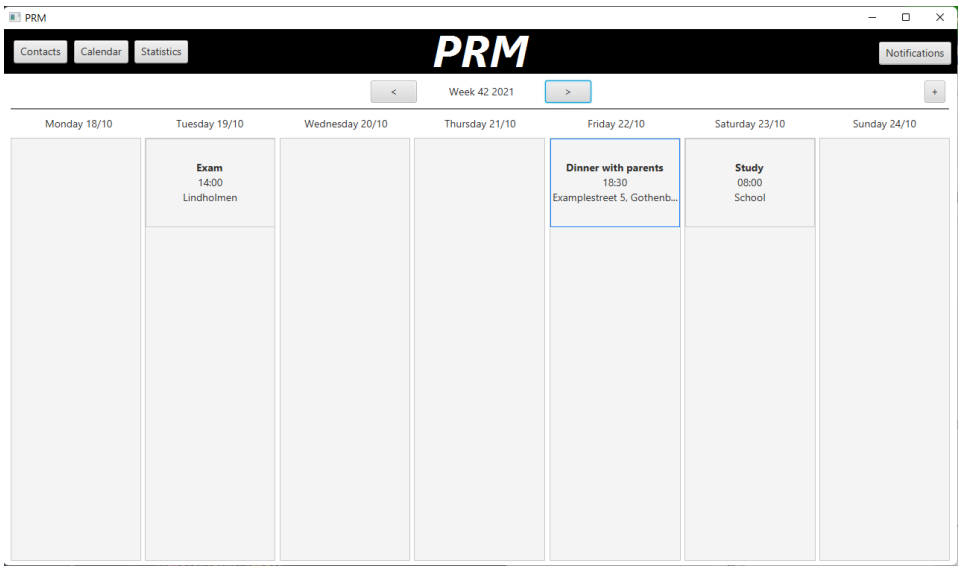

Figure 13: The final implementation of the calendar page.

From here one can use the buttons to navigate between weeks, add an event by clicking on the plus button in the top right, or edit an event by clicking on it in the calendar.

When clicking on an event or the button to add an event, the user is presented with the event creation/editing view seen below in figure 14. If the user has clicked on an event, the view will load with that event's details. If the user clicks on the plus button, the view will load with the default event details.

To navigate out of this view, one can click on the gray surroundings of the card, the save button, or the close button. Using the save button update/creates the event, while the other options exit the wiew without changing/creating the event

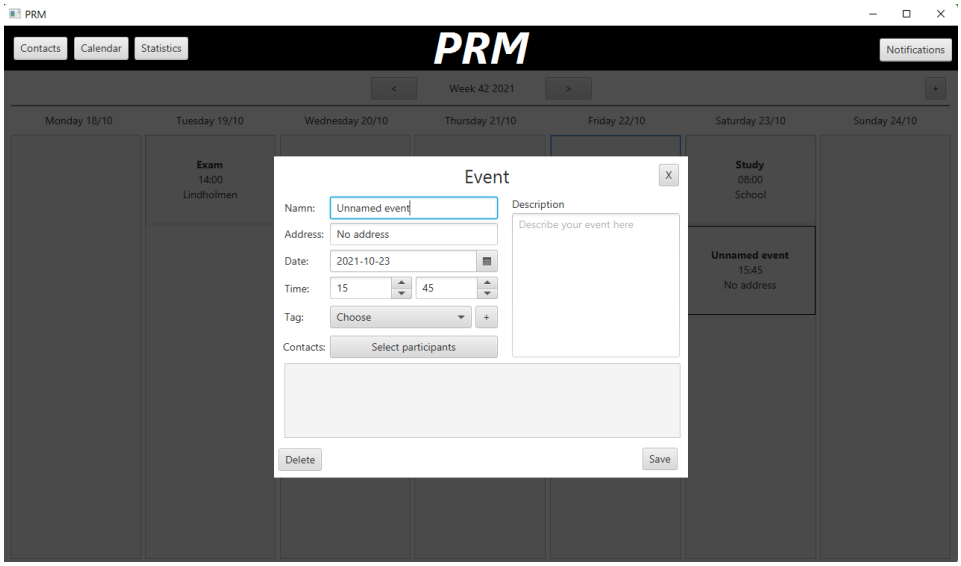

Figure 14: The final implementation of the event creation/editing view.

When adding or editing an event, one can select participants through the dialog shown in figure 15.

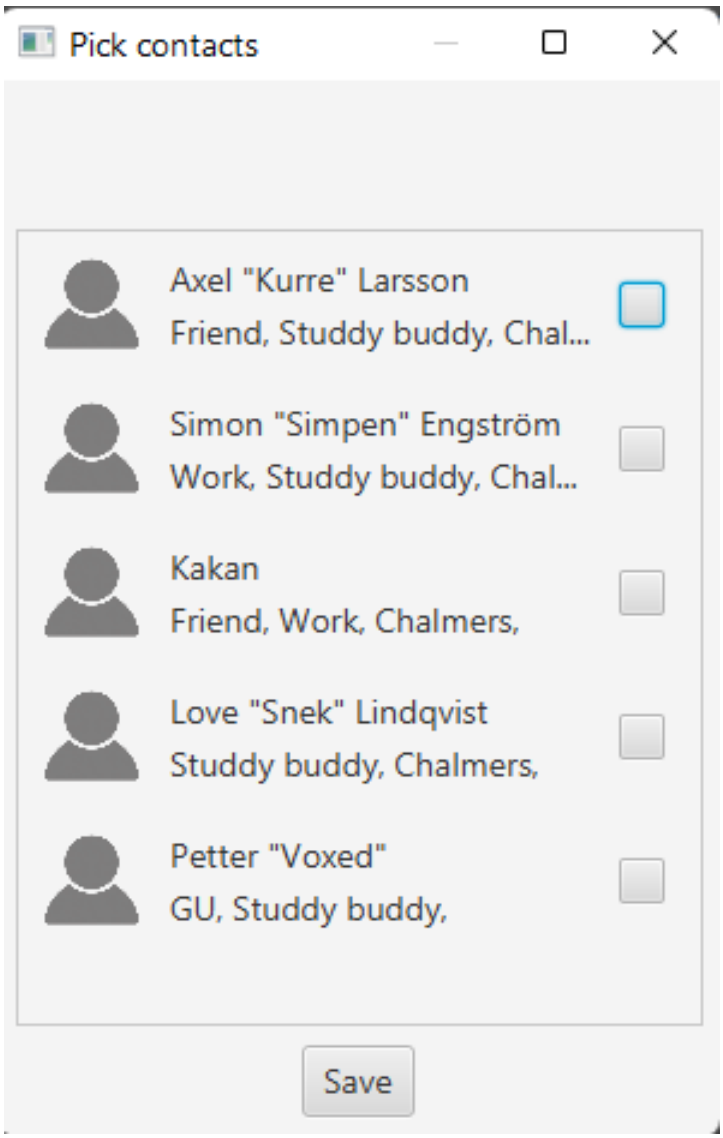

Figure 15: Final implementation of the contact-picking dialog for selecting participants on events.

On the statistics page shown in figure 16, the user is met with a pie-chart displaying the delegation of events in different categories.

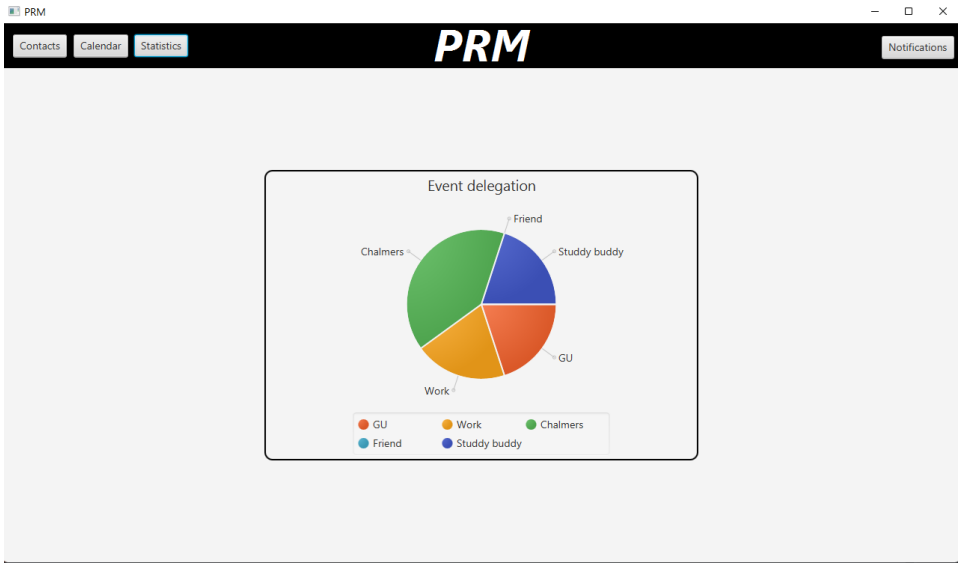

Figure 16: Final implementation of the statistics page.

On the notifications-page the user is presented with an overview of upcoming events shown below in figure 17.

Figure 17: Final implementation of the notifications page.

From both the contact-view and event edit/creation view shown in figure 11 and 14, one can choose to create a new tag by clicking on the plus button or add tag button, and filling out the dialog shown in figure 18 below. This view, along with the contact creation dialog, and the contact picking dialog are modal, meaning the user has to either close them or perform the operation in question to get back to the main interface.

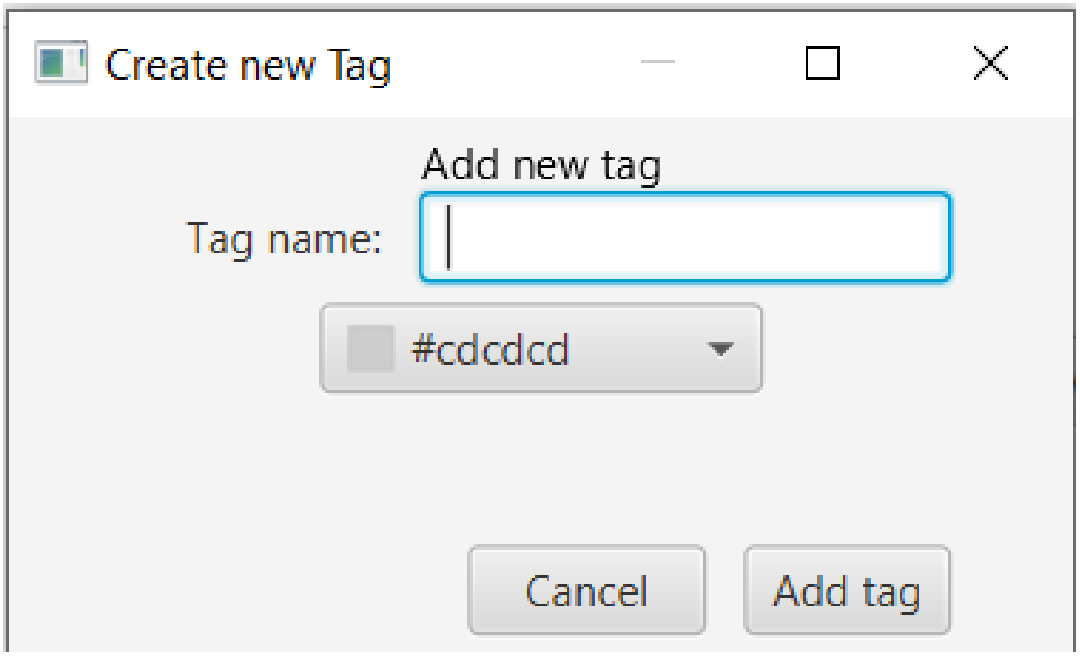

Figure 18: Final implementation of the tag-creation dialog.

3 Domain Model

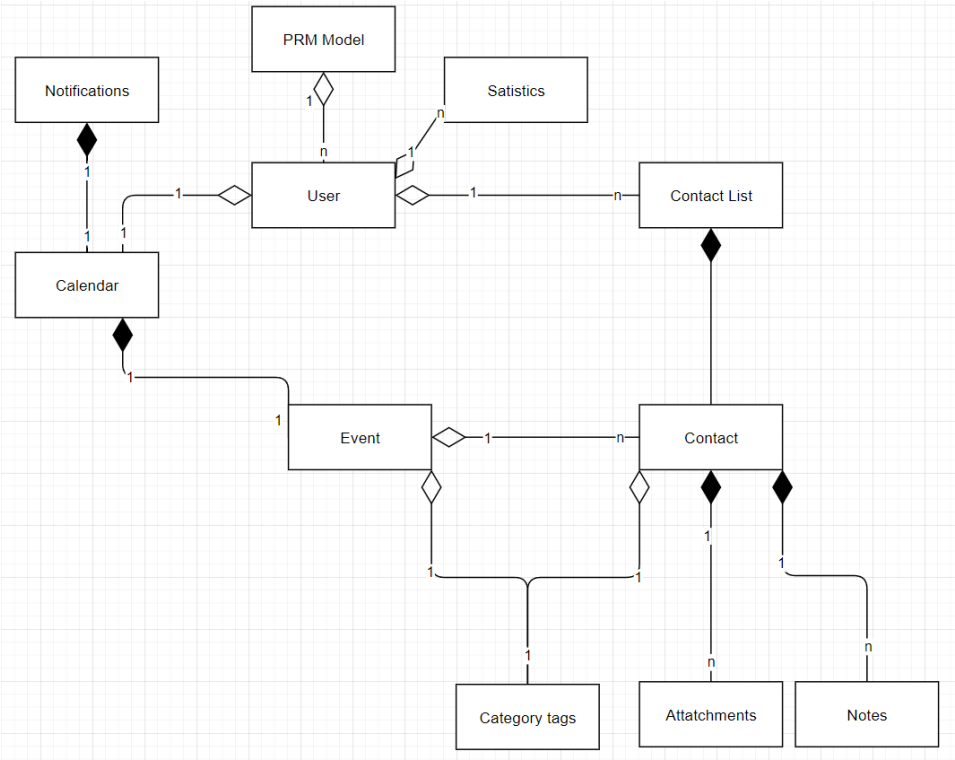

Figure 19: Domain model as of 2021-10-23

3.1 Class responsibilities

Explanation of responsibilities of classes in diagram.

- **PRM Model**: (Abstraction of the model, non-class implmenetation)
- **User**: aggregates all feature aspects of the domain and represents the content of a specific user of the application.
- **Statistics**: illustrates statistical calculations based on usage of the application.
- **Calendar**: composes events in calendar form after order of occurence.
- **Event**: represents an occurence composed of details and participants.
- **Notifications**: notifies the user about upcoming events.
- **Contact**: represents an informational composition regarding the details of an individual with a relation to the user.
- **Contact List**: represents a collection of every contact that a user has.

- **Category tags**: categorises objects with a label.
- **Notes**: represents a notebook containing details and texts.
- **Attachments**: stores files related to a specific object.

4 References

- Figma - https://www.figma.com/
- JavaFX - https://openjfx.io/
- Diagrams.net - https://app.diagrams.net/

System design document for OOPSie

Petter Blomkvist, Simon Engström,
Simon Johnsson, Axel Larsson, Love Lindqvist

October 24, 2021

Contents

1 Introduction

This document attempts to describe and cover the systemic design properties of a *Personal Relations Manager* application. The application aims to perform and illustrate informational storage regarding individual relations and related events. Its core functionality is based upon the concept of a Customer Relations Manager, but differs in its user base. Where a Customer Relations Manager aims to satisfy the needs of roles such as businessmen, the Personal Relations Manager caters towards usage by the general public. Therefore, the application's features cover the ability to collect contact details, register activities and categorize these after user-specified groups of people. These interactions with the application are then statistically analyzed and demonstrated graphically to the user in order to reflect time spent on certain relations.

1.1 Definitions, acronyms, and abbreviations

- **PRM**: Personal Relations Manager.
- **User stories**: A type of feature request written from the perspective of the application user.
- **GUI**: Graphical user interface. Allows users to interact with the software using a graphical interface.
- **IDE**: An integrated development environment.
- **MVC Architectural pattern**: A design pattern which specifies that an application consists of a data model, presentation information and control information.
- **Observer design pattern**: A design pattern which defines a subscription mechanism that notifies multiple objects about events.
- **Serialization**: The process of converting application objects to storable formats, like text files or database entries.
- **Deserialization**: The process of converting stored formats to application objects. Essentially serialization running backwards.
- **Multithreading**: The practice of executing multiple threads concurrently.
- **JSON**: A type of file format.
- **STAN**: Structure Analysis for Java.
- **PMD**: A static source code analyzer.
- **JavaFX**: A GUI library for java.
- **IntelliJ**: An IDE tailored for Java.

2 System architecture

The PRM is a standalone application, utilizing only the host file system for persistence of data between sessions. This is done by serializing the objects in the model through a service that creates data-packages of each object and saves them as JSON-files. When starting the application, the deserialization service starts, essentially running the serialization process backwards, instantiating the model.

The normal "flow" of the application occurs as follows:

1. The main method is called within the application class.
2. The application prompts the user to either create or load a user in the user wizard.
3. The chosen user is deserialized from the file system and instantiated. If a new user is created, the deserialization is not performed, since there is nothing to deserialize.
4. The controller is instantiated with the initiated model, which in turn loads the view by loading the FXML files from the view package.
5. At this stage the application is virtually idle, and will wait for the user to interact with it by for example: navigating or editing some object. When this happens, the controller class will register an event and handle it by updating the model.
6. When closing the application, the application will serialize itself by calling the serialization service.

A rough visualization of the flow can be seen in figure 1 below.

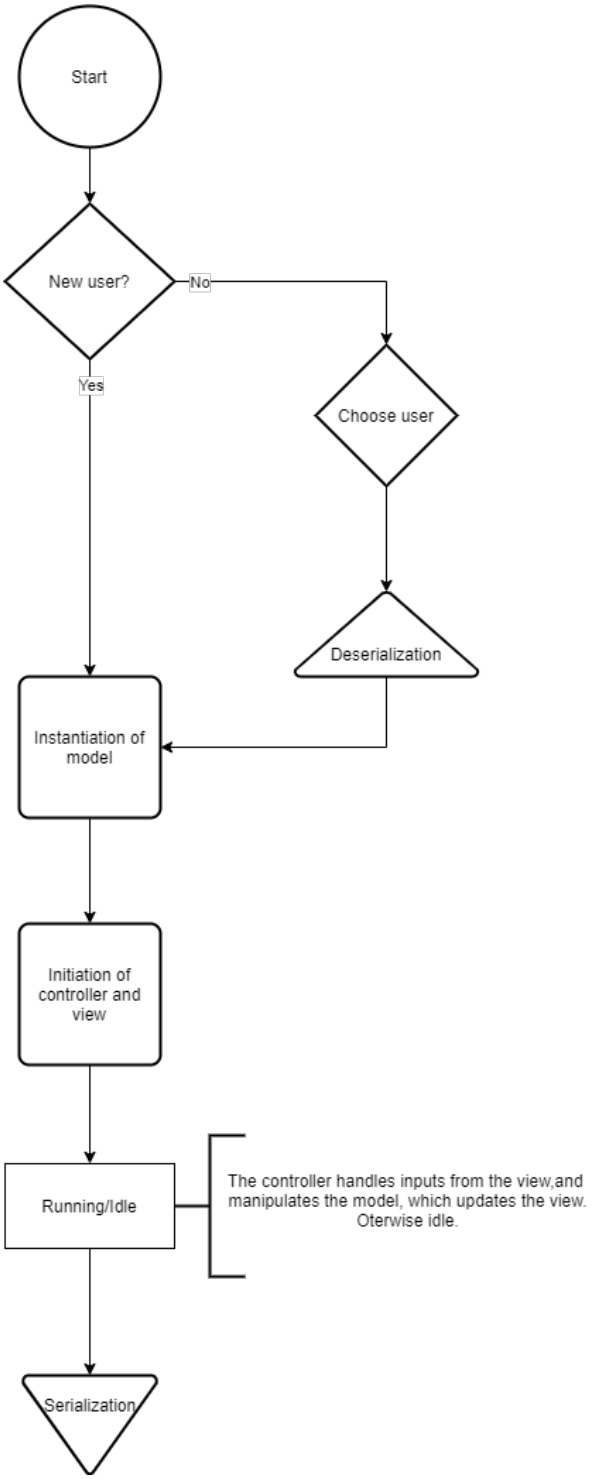

Figure 1: A flowchart representing the flow of the application.

3 System design

Figure 2 shows a top level overview for the system. The primary components consist of the model, view and service packages, while the overview contains the mediating controller. Pages are created by the *PageFactory*, and contain controller logic and an *AnchorPane* with the view.

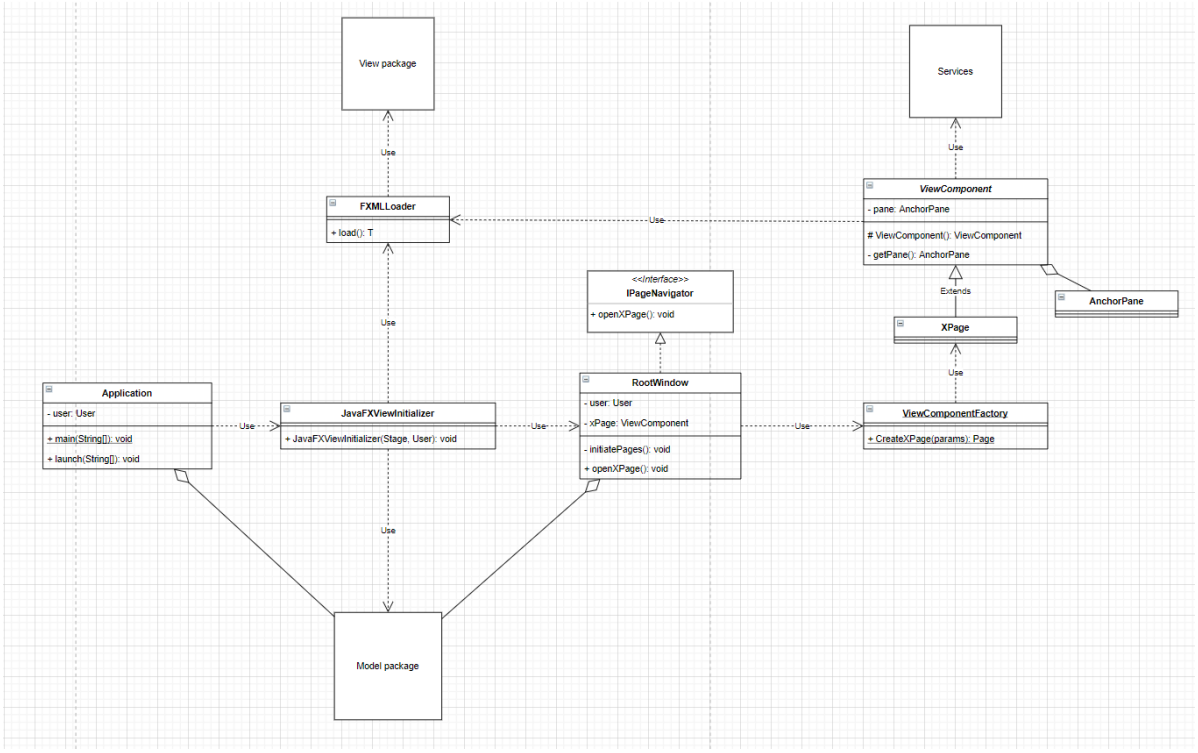

Figure 2: Overview of the MVCA pattern

Every FXML file is part of the view, and are loaded in to the program via the *FXMLLoader*, which ties the view to an instance of a controller class. Every controller is given the model-objects from it's parent, to allow it to display and operate on the model. This makes most Controllers use the most parts of the model, but prevents controllers from accessing unnecessary features from the *model*. Due to how JavaFX works, each FXML file is bound to its controller, which in turn updates/manipulates it, making the *View* and *Controller* dependent on each other. The class diagram for the *View* in this case would just be a Package of FXML files, that are not bound to one another (beyond containing FX-components that have their own controllers via JavaFX).

The *Controller* receives inputs from the user, and either updates the *View* directly, or makes changes to the objects contained in the *Model*, which then updates the *View* via

an Observer Pattern.

Figure 3 shows an overview of the design model, where *User* exists at the top, containing a *TagHandler*, a *ContactList*, and an *EventList*. The list classes hold instances of their respective objects, which both can contain one or more instances of *Tag*. The *Contact* can also contain other objects, such as *Notes*. Figure 4 shows the domain model, where an *User* is shows to have contacts which can have *System Tags*. *User* also has a *Calendar*, which contains events. This is not representative of the actual classes, but rather the method of displaying events. The contacts is also shown to have *Attatchments* and *Notes*.

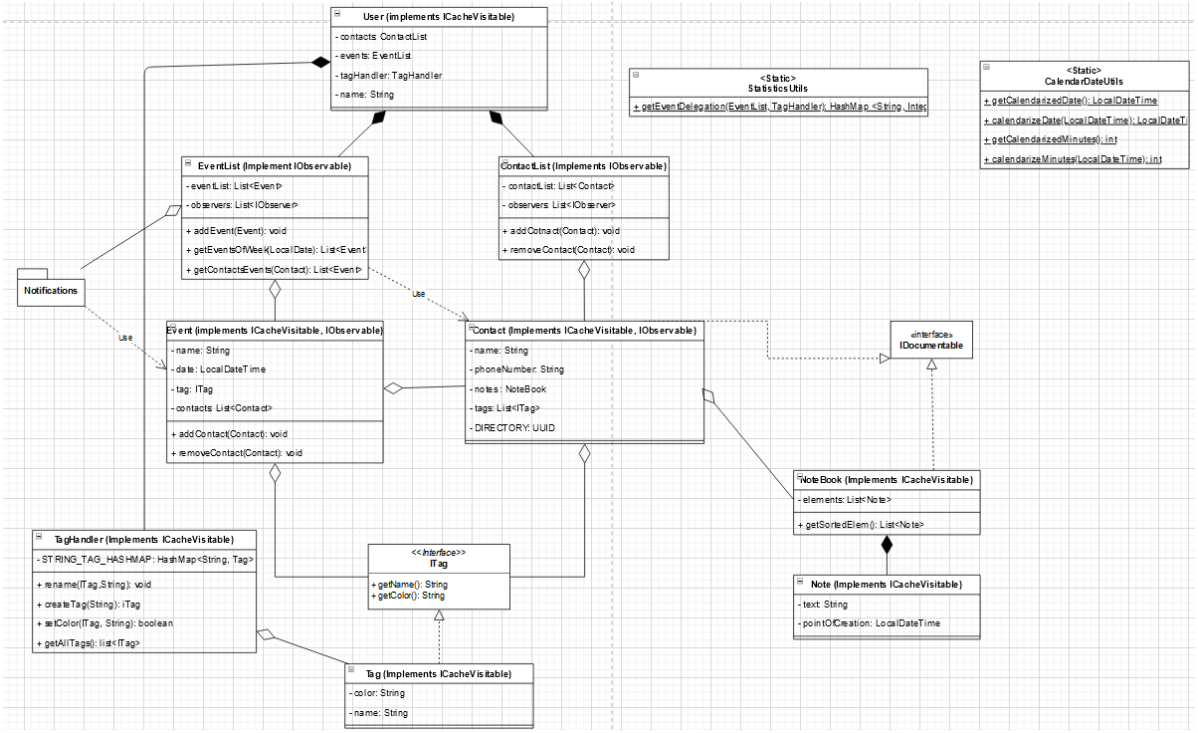

Figure 3: Overview of the design model

Two notable differences between the design model and the domain model is the lack of classes for *Calendar* and *Statistics*, which both appear in Figure 4. In the code, there is no real need for these classes, as they are just a composition of data from instances of other classes. The class which is most closely related to *Calendar* is *EventList*, as they are both compositions of *Event*. Actually grouping multiple instances of *Event* does not create a *Calendar*; *Calendar* is only a visual representation of the data. A similar explanation can be made for *Statistics*. In order to allow the user information to be stored in the database, a visitor interface (*ICacheVisitable*) is implemented by every concrete class in the model.

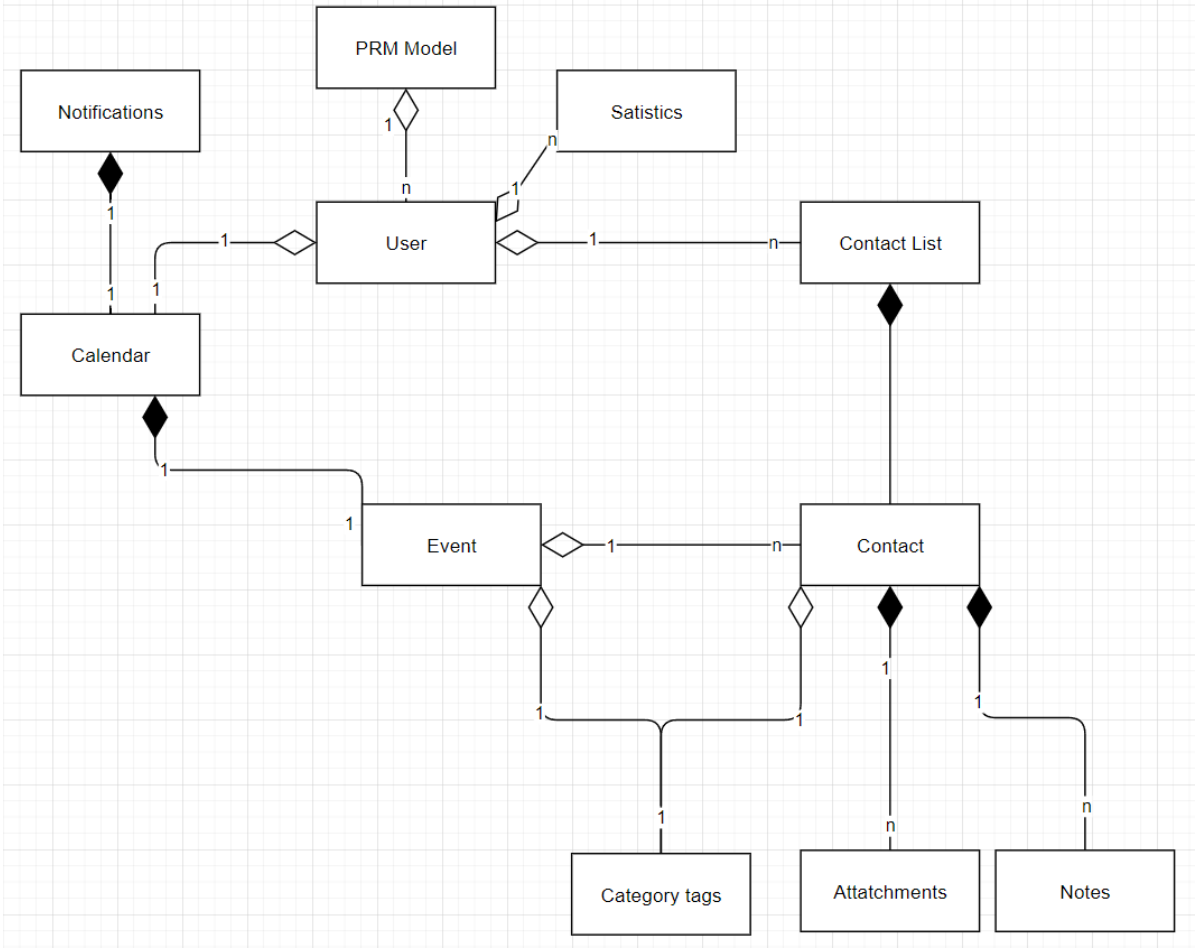

Figure 4: Overview of the domain model

Figure 5 displays an overview of the part of the model contained within the *Notifications* package, which takes in something of type chronological and broadcasts to it's listeners when the time on the object is within a certain bound. The interface *IChronological* is used to identify an implementer with the functionality required to use the package.

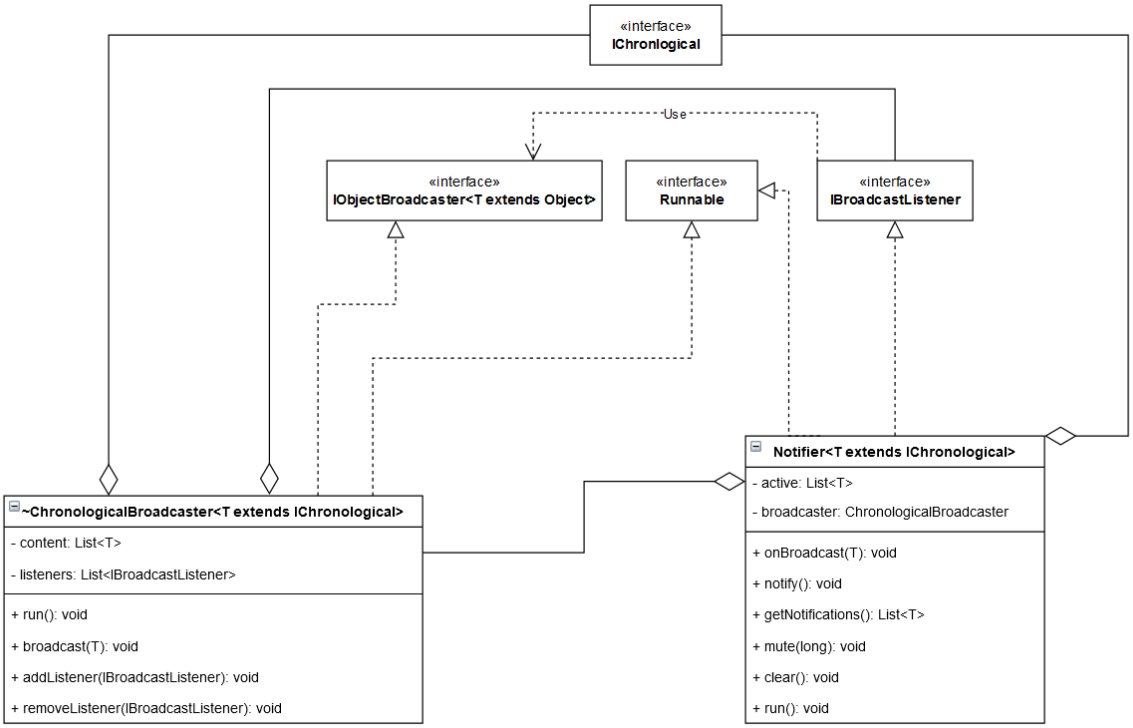

Figure 5: overview of the Notification package

Figure 6 displays an overview of the search package contained in the model, which allows for relevancy search by using other parts of the model. The search algorithm is makes use of the Damerau-Levenshtein distance algorithm. *ISearchable* describes an object which provides iterable information.

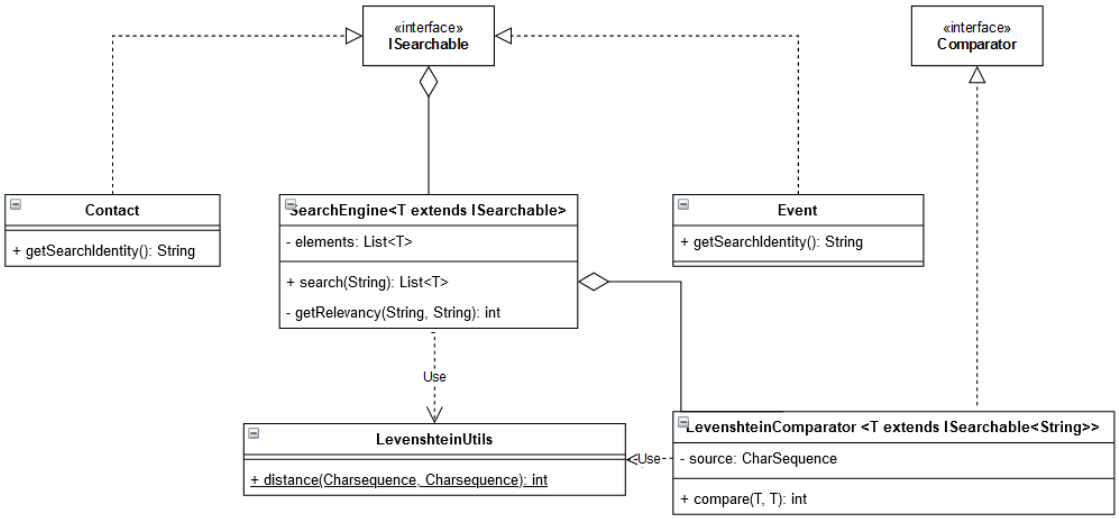

Figure 6: The Search package

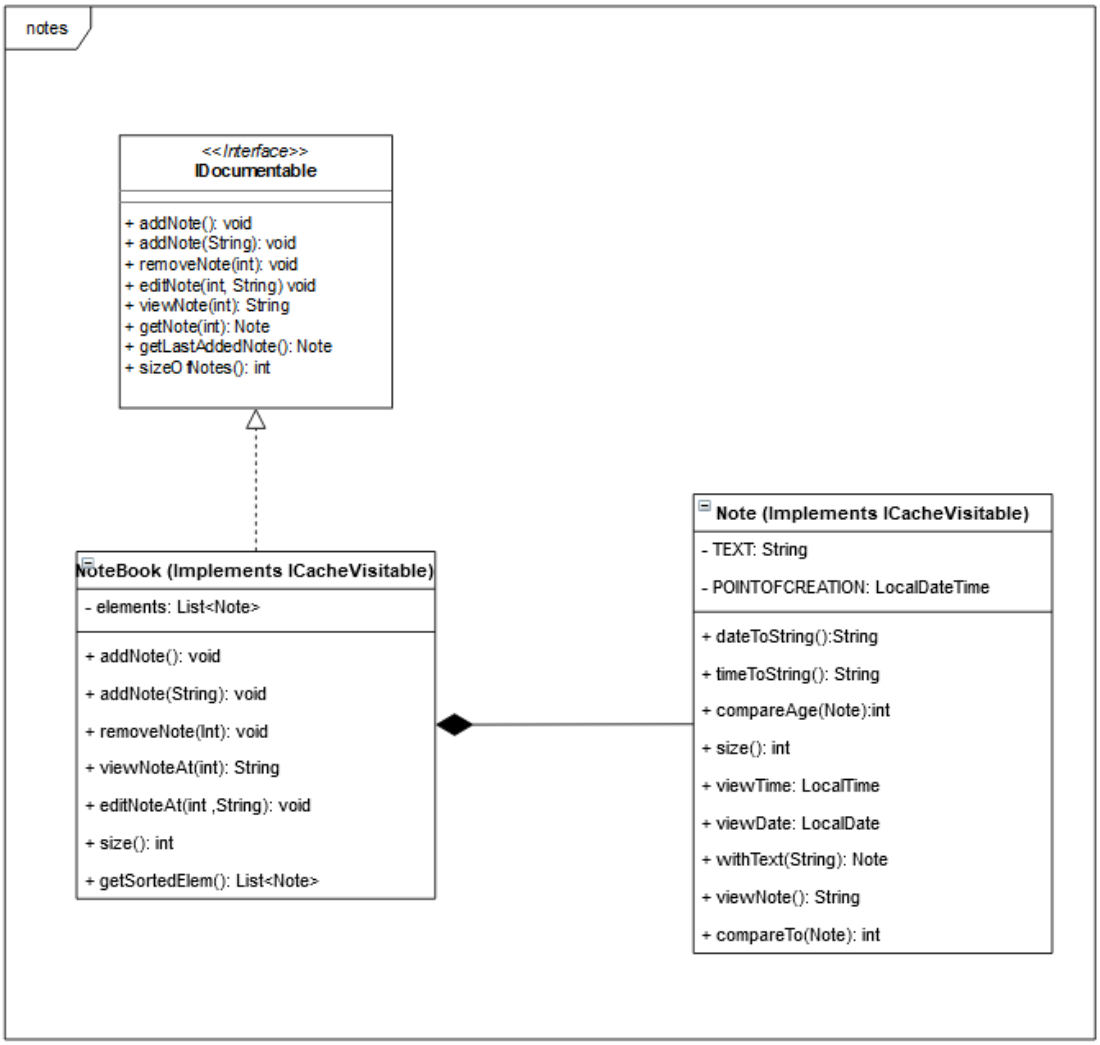

Figure 7: Caption

Figure 8 shows an overview of the VCF service, which takes in a *Path* to a file, and a *ContactList*. The service then reads *vcf files and adds them to the list of contacts.

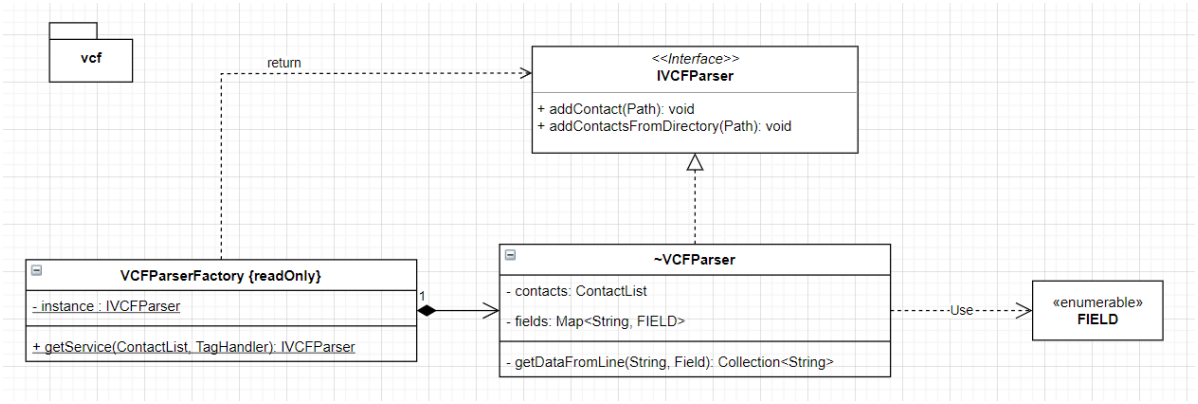

Figure 8: Overview of the vcf-parser package

Figure 9 shows an overview of the attatchment service, which allows a user to save files like images pertaining to a specific contact.

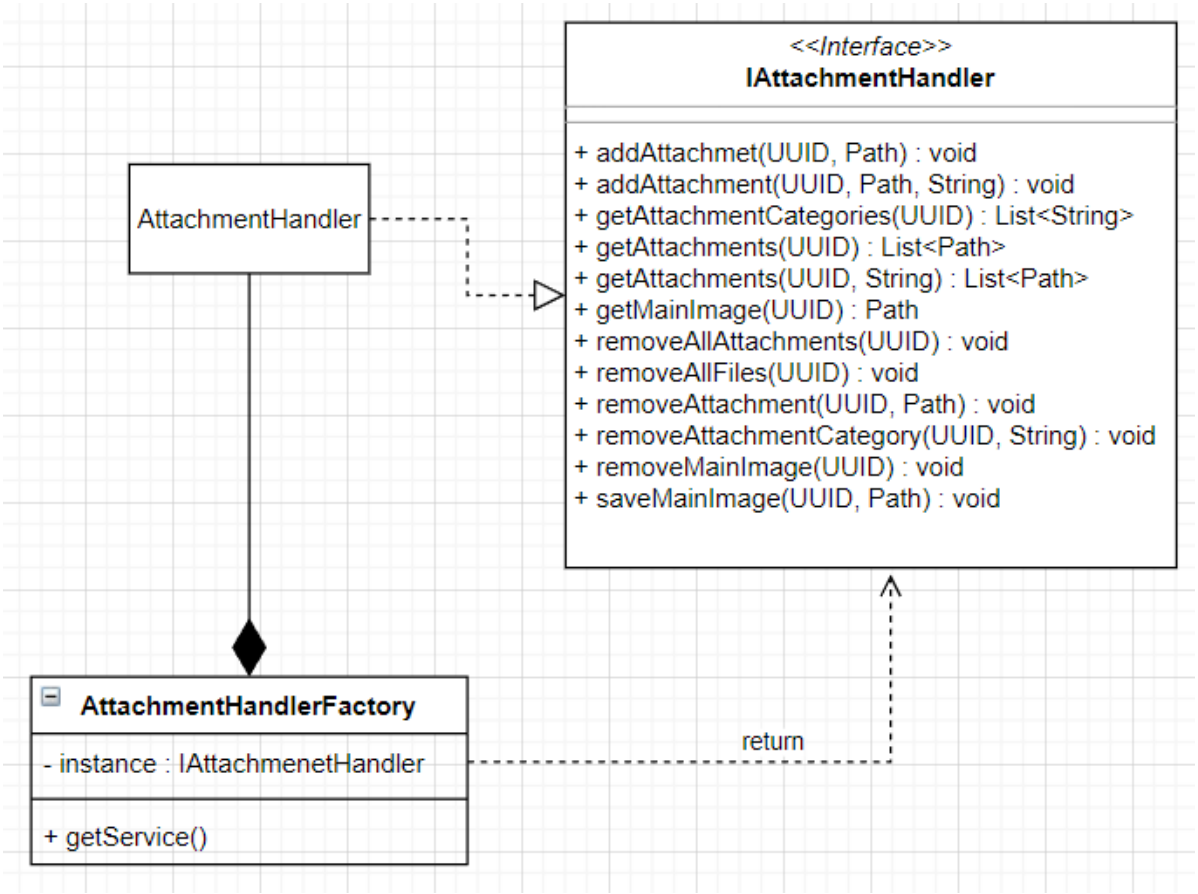

Figure 9: Overview of the attatchments-package

Figure 10 shows an overview of the database service, which facilitates the saving and loading of users to the hard-drive. While the *Database* class is not abstract, all of its logic is moved to separate abstractions such as *IDatabaseLoader* which controls all the user loading logic.

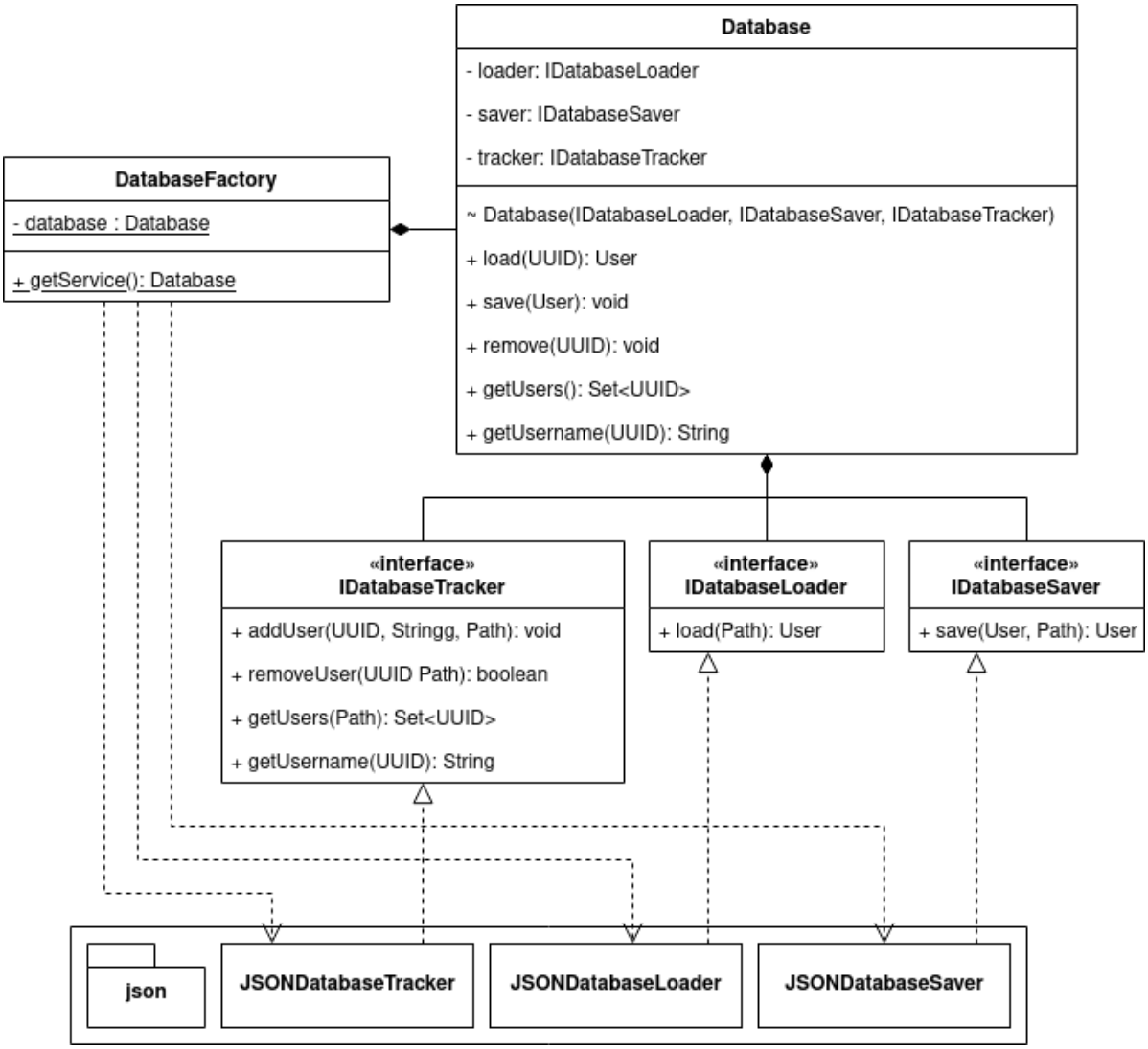

Figure 10: Overview of the database package

3.1 Design Patterns

A lot of classes implements the visitor pattern to allow data to be extracted for database serialization, as well as constructors that accept the inner class *ClassName.ClassNameCache*. This allows services like the database and *VCFParser* to create objects with specific information.

As stated before the *Observer Pattern* is used to allow the view/controller to subscribe to parts of the model, and be alerted about an update in the model package.

The *Singleton Pattern* is used for factories and services that only need one instance.

The *Factory pattern* is used to statically instance objects as abstract types without revealing their construction to the client. E.g. the *Page Factory* class

The *Bridge pattern* is used to allow for an object to contain different implementations of the same abstract type. This prevents the need to change the object's internal hierarchy when an alternative implementation is required. E.g. the *Levenshtein Comparator class* which allows for any implementation of CharSeuqence to be used.

The *Chain of responsibility pattern* is used when multi-threading objects to allow for the upmost client layer to instance and run the thread. E.g. the *Notifier* class which passes along its run method to the internal runnable object.

4 Persistent data management

Persistent data storage consists of two different services, the database service and the attachment handler service. All persistent data is stored in a .prm directory which will be created in the user home directory, e.g. "C:\Users\<username >\.prm" on windows systems and "/home/<username>/.prm" on unix based systems.

The database service takes care of storing and loading the user model to the hard drive. It does this by serializing the entire user model to the JSON format which is then saved to a file in the .prm/user directory with the users UUID as filename. The serialization is done by first converting the entire user model into a hierarchy of less complex java objects referred to as "records", this record hierarchy is then converted into text using the Google maintained library GSON. Loading the model works in a very similar way, first the JSON data is converted into the record hierarchy using GSON, then this hierarchy is converted back into the real user model. The code that takes care of converting to and from the record hierarchy makes heavy use of the visitor pattern. In order to track which users exists and their names there also exists a tracker file located at .prm/db. This file contains the user UUIDs along with their names.

The attachment handler service allows the application to use IDs to store attachments and one main image for different entities in in the program. The attachments are stored in .prm/{id}/attachments and can be stored in different subcategories which are in fact subdirectories of the attachments directory. The main image is stored in

.prm/{id}/mainImage and only allows one image to be stored at a time. These files do not need to be serialized in any way as they are not objects in run-time but simply files copied to the data folder for the PRM application.

5 Quality

5.1 Testing:

The functionality of the application is asserted by Unit Testing using the junit-4 package. The test directory structure mirrors the programs structure. As an example the Contact class is located in `./src/main/java/model/Contact` and the test code in `./src/test/java/model/ContactTest`. Continuous integration is handled through the service Travis and is found under this Build History.

5.2 STAN

STAN uses the compiled class files to generate a dependency diagram.

5.2.1 Toplevel

The red dependencies in figure 11 between controller and application comes from the HostServiceProvider class that provides HostServices so that the controllers can run OS level commands for opening files and links.

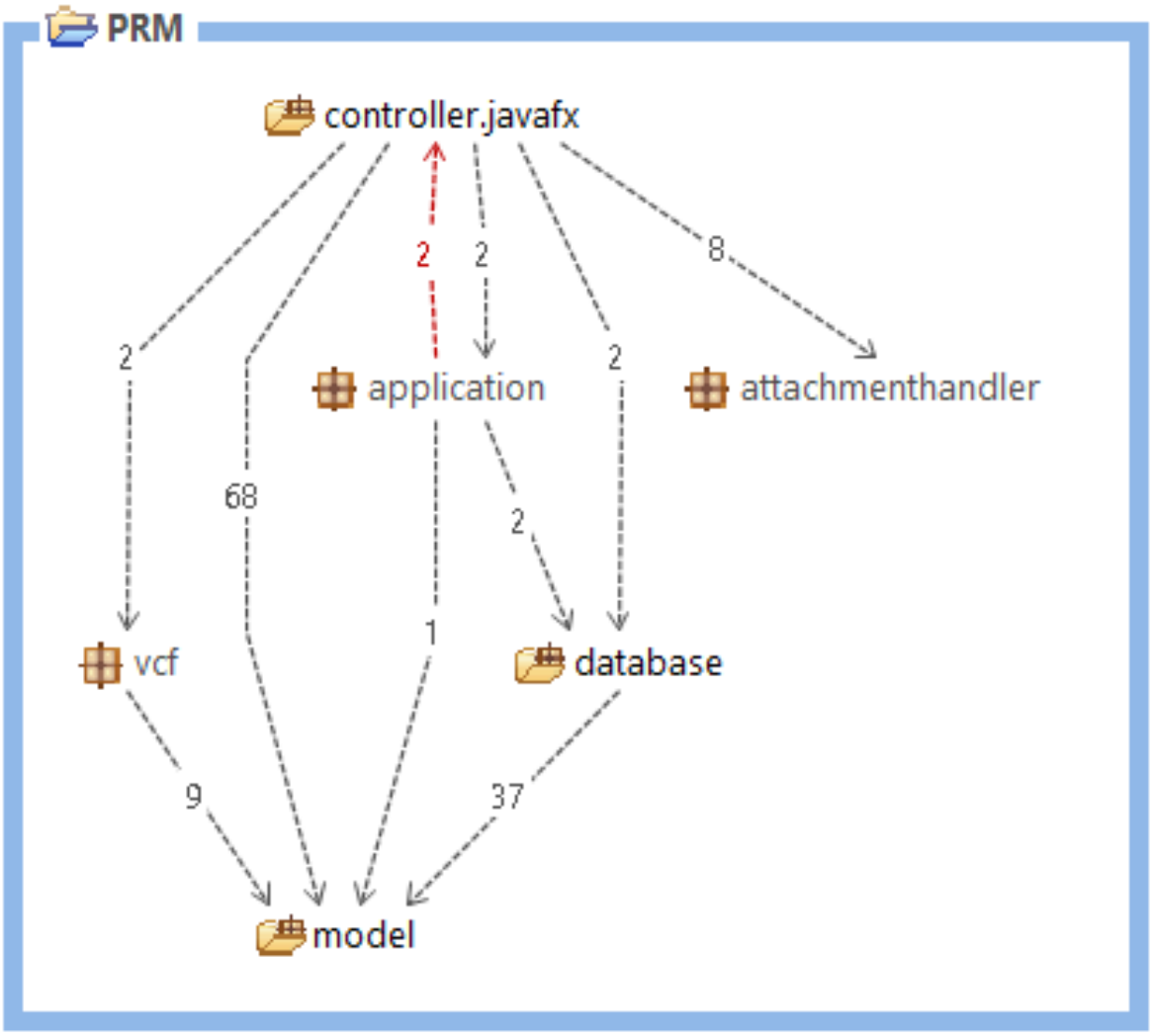

Figure 11: A top level view of the compiled code with packages collapsed.

5.2.2 Model

The reason for the red arrows in figures 12 and 13 is because the model is quite tightly coupled between sub-packages. This is due to the fact that the packages are mainly there for organizational reasons.

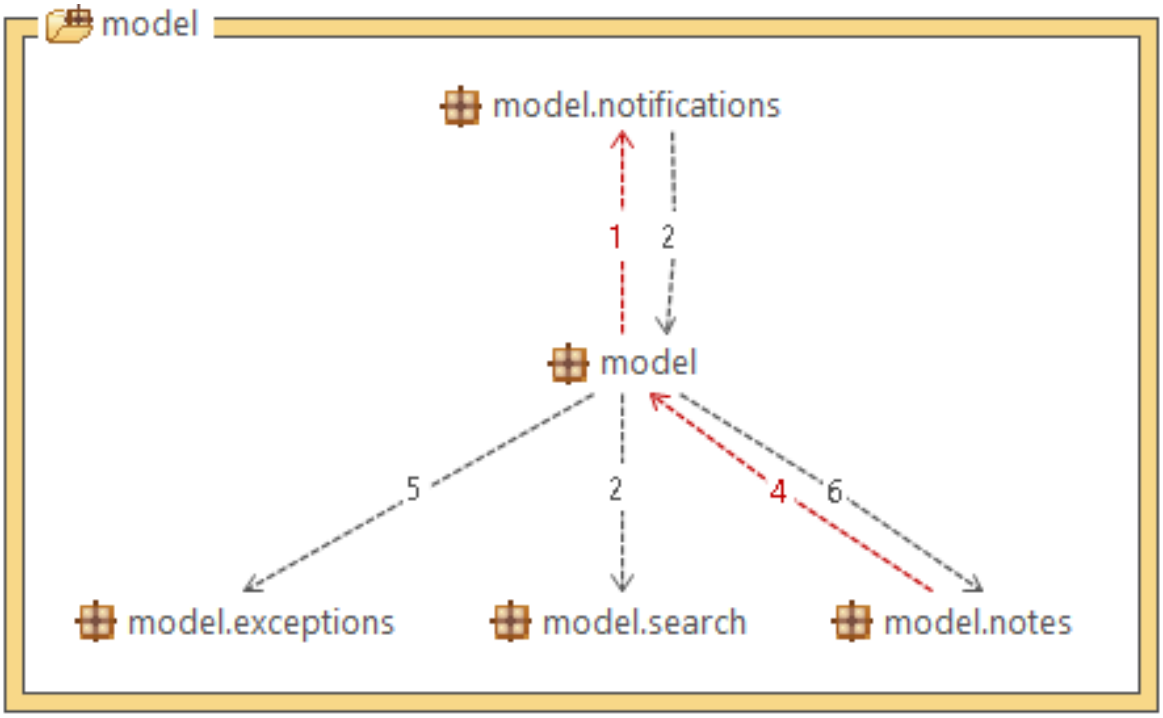

Figure 12: A collapsed view over the model package.

Figure 13: An expanded view over the model package.

5.3 Controller

The red in figure 14 is once again due to organization. Everything there is related to the JavaFX controller but the components are all put into a subdirectory to make it easier to navigate.

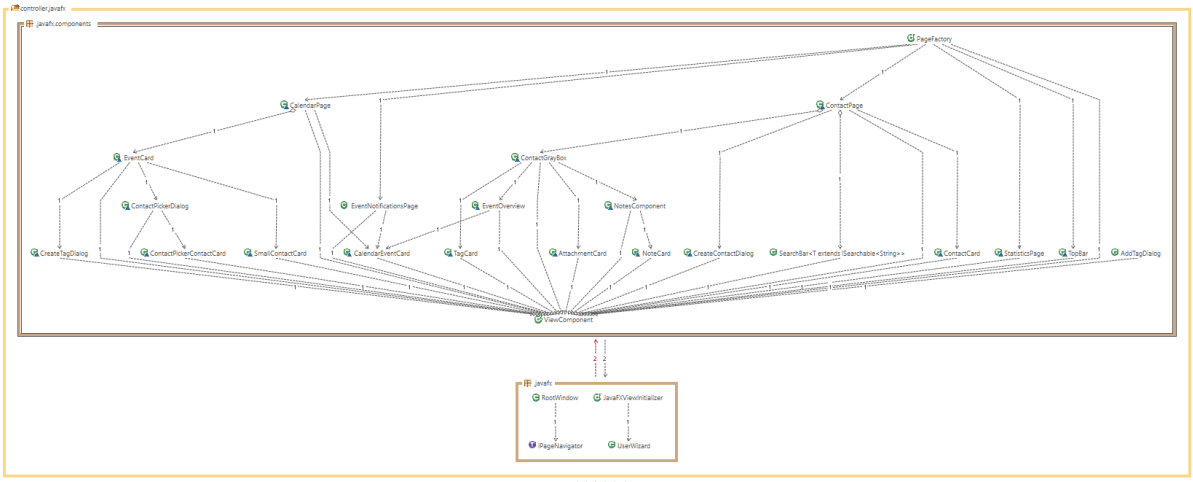

Figure 14: An expanded view over the controller package.

5.3.1 Services

The only red arrow in 15 is because the DatabaseFactory class provides an instance of the JSON implementation.

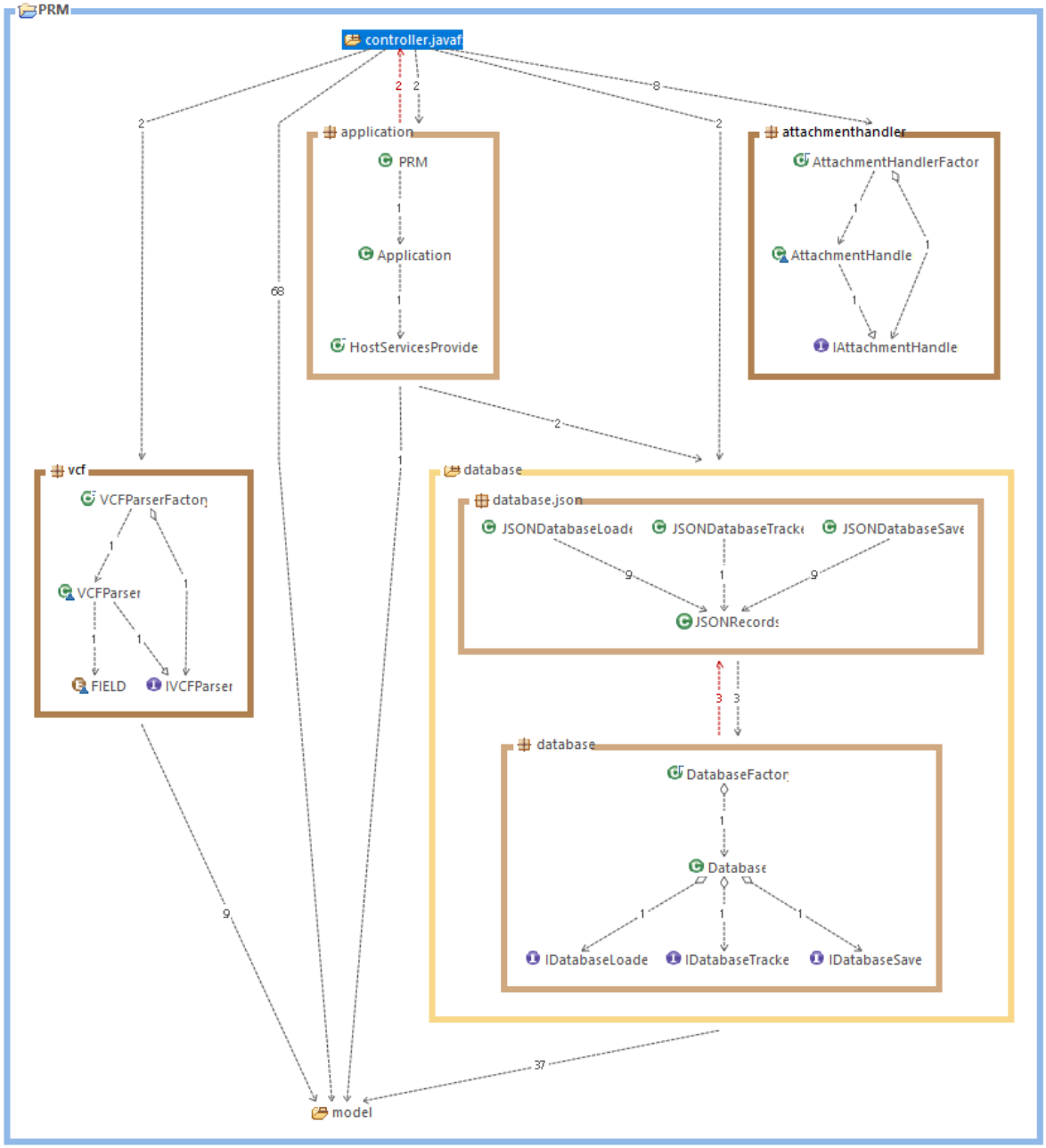

Figure 15: A toplevel view with services and application packages expanded.

5.4 PMD

The provided quickstart java ruleset was used for running all the tests. The subsections below include the whole output of PMD but split into different subsections for packages. A recurring complaint is unused import when using a wildcard import. These complaints were not addressed as the amount of import statements drasti-

cally increase if we import everything individually. IntelliJ import optimization also chooses to use wildcard imports when enough methods from the same library are used.

5.4.1 Application

HostServicesProvider.java:8: ClassNamingConventions: The utility class name 'HostServicesProvider' doesn't match '[A-Z][a-zA-Z0-9]+(Utils?—Helper—Constants)'
PRM.java:3: UseUtilityClass: All methods are static. Consider using a utility class instead. Alternatively, you could add a private constructor or make the class abstract to silence this warning.

5.4.2 JavaFX controller

JavaFXViewInitializer.java:14: ClassNamingConventions: The utility class name 'JavaFXViewInitializer' doesn't match '[A-Z][a-zA-Z0-9]+(Utils?—Helper—Constants)'
components/ContactGrayBox.java:19: UnnecessaryImport: Unused import 'model.*'
components/ContactPage.java:10: UnnecessaryImport: Unused import 'model.*'
components/EventCard.java:7: UnnecessaryImport: Unused import 'javafx.scene.control.*'
components/EventCard.java:11: UnnecessaryImport: Unused import 'model.*'
components/PageFactory.java:11: ClassNamingConventions: The utility class name 'PageFactory' doesn't match '[A-Z][a-zA-Z0-9]+(Utils?—Helper—Constants)'
components/StatisticsPage.java:7: UnnecessaryImport: Unused import 'model.*'

There are complaints that have been left out here to avoid getting a very large wall of text. The removed complaints were all UnusedFormalParameter complaints. While these parameters are unused within the methods themselves they cannot be removed because JavaFX will only allow a method to be set as onAction for a button if it takes an ActionEvent as a parameter.

5.4.3 Services

attachmenthandler/AttachmentHandlerFactory.java:8: ClassNamingConventions: The utility class name 'AttachmentHandlerFactory' doesn't match '[A-Z][a-zA-Z0-9]+(Utils?—Helper—Constants)'
database/DatabaseFactory.java:12: ClassNamingConventions: The utility class name 'DatabaseFactory' doesn't match '[A-Z][a-zA-Z0-9]+(Utils?—Helper—Constants)'
database/json/JSONDatabaseLoader.java:5: UnnecessaryImport: Unused import 'model.*'
database/json/JSONDatabaseSaver.java:5: UnnecessaryImport: Unused import 'model.*'
vcf/VCFParser.java:90: CloseResource: Ensure that resources like this Scanner object are closed after use
vcf/VCFParserFactory.java:9: ClassNamingConventions: The utility class name 'VCFParserFactory' doesn't match '[A-Z][a-zA-Z0-9]+(Utils?—Helper—Constants)'

The scanner in VCFParser is closed when the method is done. The only way it does not get closed is if the method throws a NoSuchElement exception but this should not happen as we use hasNext on the scanner before retrieving the element. This should probably be handled with a try catch just in case but it should never cause issues even as it is currently.

5.4.4 Tests

model/ContactListTest.java:10: UnnecessaryImport: Unused import 'org.junit.Assert.*'
model/ContactTest.java:9: UnnecessaryImport: Unused import 'org.junit.Assert.*'
model/EventTest.java:13: UnnecessaryImport: Unused import 'org.junit.Assert.*'
model/ICacheVisitorTest.java:11: UnnecessaryImport: Unused import 'org.junit.Assert.*'
model/TagHandlerTest.java:11: UnnecessaryImport: Unused import 'org.junit.Assert.*'

model/UserTest.java:9: UnnecessaryImport: Unused import 'org.junit.Assert.*'

5.5 Known Issues:

- The same notification will sometimes be pushed multiple times.
- If the computers language is set to English then characters like "åäö" will break when saving on certain systems.
- The folders containing attachments and such for contacts are not removed after the contact is removed.
- There is an option to choose opacity of a tag even though it is not supported.
- Manually typing the date in events does not change, the calendar interface must be used.
- Creating a very long tag breaks the tag chooser in the GUI.
- Certain files added can not be removed. The only case currently known is if you add a temporary file from Firefox as an attachment then try to remove it. Possibly caused by some protected flags on the file or something similar.
- UI does not scale very well.

6 References

Libraries

- javafx - https://openjfx.io/
- javafxml - https://docs.oracle.com/javafx/2/get_started/fxml_tutorial.htm
- junit - https://junit.org/junit5/
- gson - https://github.com/google/gson
- STAN - http://stan4j.com/
- PMD - https://pmd.github.io/
- Travis - https://www.travis-ci.com/

Axel Larsson, Simon Engström, Love Lindqvist, Simon Johnsson, Petter Blomkvist

Peer review - MonOOPPoly

- Do the design and implementation follow design principles?
    - Does the project use a consistent coding style?
        - The project uses a consistent style following the standard styling for Java. There are some minor inconsistencies where a blank line or space is missing so it would be good to format all files using IntelliJ.
    - Is the code reusable?
        - Generally the code is not very reusable due to it being adapted to the usage in the game of monopoly, instead of being interesting by itself.
        - An example of this is the dice class which is very specific to rolling two dice. This makes some methods irrelevant in scenarios where only one, or more than two dice are needed. Consider creating a single class representing a die and make a collection class consisting of a selectable amount of die.
        - Another example of a class that would be hard to reuse is the property class. It is an anemic class and the Player class has to handle all the logic for purchasing a property. If this functionality was in the property instead it would be easier to purchase properties in other parts of the code as well.
        - Some methods are pretty large, especially in the Game class. Functional decomposition could improve code reusability within classes.
    - Is it easy to maintain?
        - BoardController handles a lot of logic that would be more appropriate in the model. For example there is a very large switch statement for converting a board position to x and y coordinates on the grid. This should probably just be a hashmap in the board class if it needs to be hardcoded or alternatively calculated dynamically.
        - All jail logic is currently in the game class which makes it hard to change, it would probably be more easily maintainable if it was delegated to a class.
        - The package structure is very well done and easy to understand.
    - Can we easily add/remove functionality?
        - It could be hard to implement new functionality in the space subtype hierarchy since a lot of dynamic type checking is performed. It could require you to extend these statements throughout the application when adding new types.
    - Are design patterns used?
        - A model-view-controller structure is used.
- Is the code documented?
    - Some things are documented but a lot of public methods are lacking documentation.
- Are proper names used?
    - The code mostly follows the standard regarding naming.

    - Package names should be lowercase to avoid naming conflicts with classes.
    - Locale can be misleading as it is usually used for defining the user's language in software. The method for getting the color is called getSectionColor so a reasonable refactor would be renaming the class to Section.
- Is the design modular? Are there any unnecessary dependencies?
    - The code is generally modular as it is divided into classes and packages. Some methods are large and could be divided for more modularity.
    - An example of less modular code is the Player class. It has an integer to keep track of turns in jail. This integer only has a value and getter/setter and the logic is handled in other parts of the program. This means that if you want to use the player on a gameboard that does not have a jail, there is dead code in the Player class. The player would be a more well defined module if the turns in jail integer was in the same place as the jail logic.
    - JavaFX color should not be in the model as it makes the model depend on the view. Instead the model could use a hex code.
- Does the code use proper abstractions?
    - Abstraction is used in the program but it is not implemented in a beneficial way. In the Game class the type of currentSpace is Space, the superclass to all types of spaces on the board. This hides all functionality of the subclasses that is not inherited from Space. However, the only functionality that the Space class has is to getSpaceName which means that the instanceof keyword and type casting has to be constantly used in the Game class to use the functionality of the subclasses.
- Is the code well tested?
    - The model has a very high method coverage, only two classes don’t have 100% line coverage. This is great but there are some quality issues.
        - Some tests like buyPropertyTest are large with many asserts and should probably be split up. The BoardTest only tests if one certain space is present, but nothing else.
- Are there any security problems, are there any performance issues?
    - No major security or performance issues were found, which is great.
- Is the code easy to understand? Does it have an MVC structure, and is the model isolated from the other parts?
    - The given UML diagrams made it very easy to get an understanding of how the program works. The fact that the classes are modeled after the monopoly game also makes it easier to understand.
    - The model is not very well encapsulated, all classes and constructors are public which exposes it to the whole program. Consider making all classes package-private if they do not have a reason to be public.
    - The code follows the MVC pattern, but as mentioned above the model does depend on the view through javaFX color which is not great.
- Miscellaneous improvements
    - Could use observers to update the view.
    - The JavaFX dependency is not added through maven, making it tedious to build the application. This can be fixed by adding the build plugin and build dependencies in pom.xml.