



Alarm Clock: Technische Dokumentation

Jonathan Hyams
Pascal Schmalz

16. Juni 2017

Inhaltsverzeichnis

1	Zweck des Dokument	4
2	Architektur	4
2.1	Auslösen einer Notification	6
2.2	Konfiguration	6
3	Main	6
3.1	Class: Main	6
4	Controller	7
4.1	Class: Controller	7
4.2	Class: DateTimePicker	8
5	Model	8
5.1	Class: Model	8
5.2	Interface: DataBaseAdapter	9
5.3	Class: BinaryDBAdapter	9
5.4	Reminder	9
5.5	ReminderList	10
5.6	Specific Reminder	10
5.7	Poller	10
5.8	ConfigReader	10
5.9	Konfiguration	11
6	Filtering	11
6.1	Architektur	11
6.2	Higher Order Functions	11
6.3	echte Higher Order Functions in Java	12
6.4	Code Beispiel	12
6.5	Implementierungen von Filtern	12
6.5.1	hasTag	13
6.5.2	IsInNextMin	13
6.5.3	IsInNextSecond	13
7	Notification	13
7.1	Interface: Notification	13
7.2	Class: ConsoleNotification	14
7.3	Class: JavaFxNotification	14
7.4	Class: MultiReminderNotification	15
7.5	Class: NotificationHandler	15
7.5.1	Methode showPastEvents	16
7.6	IsThisYear	17
7.7	IsThisMonth	18

7.8	IsToday	18
8	Resources	18
8.1	Resources	18
9	Tests	19
10	Tools	19
11	Versionskontrolle	20

1 Zweck des Dokument

Dieses Dokument dient für die Leute, die unseren Code analysieren, verstehen und bewerten wollen. Wir erklären hier, wie unser Programm strukturiert ist, wie es funktioniert und warum wir einige Entscheidungen getroffen haben.

2 Architektur

Unse Programm implementiert ein MVC Pattern. Die Komponenten Model und Controller sind als Java Pakete schnell ersichtlich. Die View wird aber nicht in einem Java-Paket ausprogramiert. Da wir JavaFx neutzen, wird die View durch JavaFx definiert. Die Implementierungsdetails findet man unter resources/mainWindow.fxml. Auch das Notification Packet könnte man zur View zählen.

Die wichtigsten Komponenten sieht man in der Abbildung Systemübersicht.

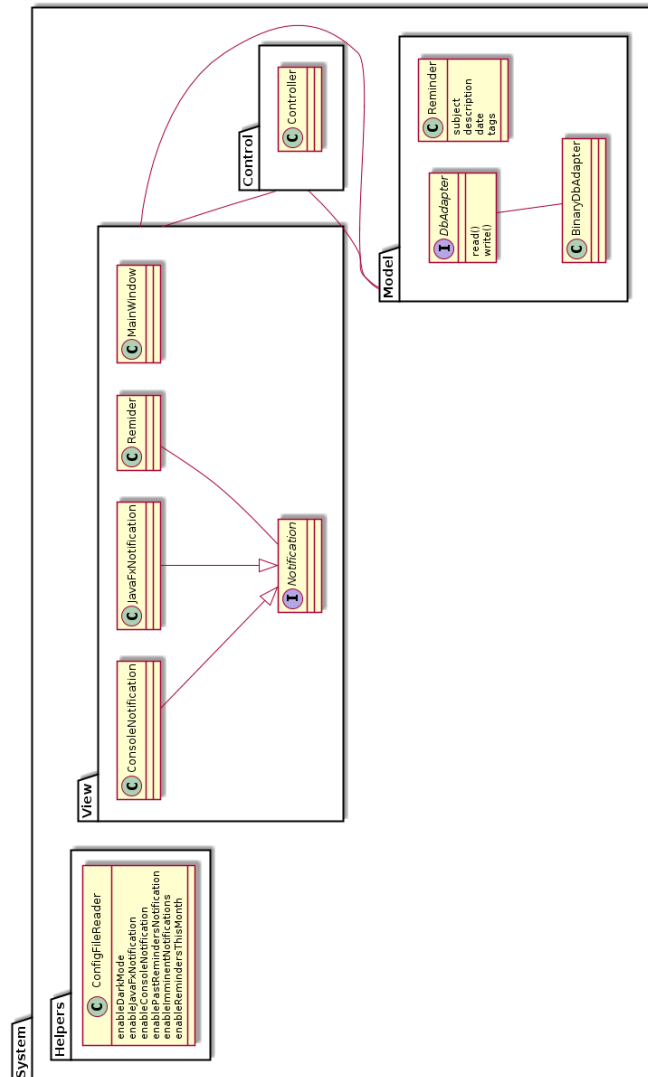


Abbildung 1: Systemübersicht

Spezielle Aufmerksamkeit bedarf das Auslösen eines Reminders.

2.1 Auslösen einer Notification

Der Poller stösst regelmässig den NotificationHandler an. In diesem werden Kriterien definiert, nach welchen die Reminders gefiltert werden. Zum Beispiel alle Reminders, welche in der nächsten Stunde beginnen.

Dann wird über alle Reminders iteriert. Jedem Reminder, wird dabei der Filter an die NotifyIf() Funktion übergeben. Der Reminder testet nun anhand des Filters selbstständig, ob das Kriterium zutrifft oder nicht. Dazu nutzt er seine meetsCriteria() Funktion. Falls diese eine positive Antwort gibt, wird die doNotify Funktion aufgerufen. Diese lädt eine Liste mit den Notifications, welche im ConfigReader definiert wird. Jeder dieser Notifications wird der Reminder übergeben, welche die Notifications auslösen will. Dann wird die Notifications abgesendet.

2.2 Konfiguration

Die Konfigurationsmöglichkeiten sind im ConfigReader File zentral gelöst. Somit wird es später auch leicht möglich, über diese Klasse ein Konfigurationsfile einzulesen, und die entsprechenden Komponenten zu konfigurieren.

3 Main

3.1 Class: Main

Dies ist die Klasse die die Main methode enthält. Sie erbt von der Klasse Application, da Sie das JavaFX-GUI launched. Application implementiert die start methode, die ein Stage als Parameter hat. Diese Stage ist das Hauptfenster mit der Tabelle von Reminders.

```
1 Platform.setImplicitExit(false);
```

Wenn Platform.setImplicitExit true ist, werden Befehle wie Platform.runlater() ignoriert, sobald alle JavaFX Fenster geschlossen sind. Das würde heissen, dass wenn das Hauptfenster geschlossen wurde (und alle Popups), würden spätere Popups nicht mehr erscheinen. Dies wollen wir verhindern, also ist es auf false gesetzt.

```
1 Parent root = FXMLLoader.load(getClass().getResource(
    windowName));
```

windowName ist der Name der fxml Datei (mainWindow.fxml), in der das Aussehen und der Controller definiert wird. "mainWindow.fxml" befindet sich im package "resources".

```
1 if (new ConfigReader().isEnabledDarkMode()) {
    scene.getStylesheets().add("dark.css");
    } else {
    scene.getStylesheets().add("styles.css");
5 }
```

Der ConfigReader kann so angepasst werden, das man hier das gewünschte CSS File bekommt.

4 Controller

4.1 Class: Controller

Die Controller Klasse arbeitet eng mit der mainWindow.fxml Datei zusammen. Würde man dieser Klasse den Namen ändern, so müsste man im fxml das Statement

```
1 fx:controller="alarmClock.controller.Controller"
```

anpassen

In der fxml Datei hat jedes Item im GUI eine ID. Damit man im Java Code auf die jeweiligen Komponenten zugreifen kann, hängt man bei der Initialisierung ein @FXML Tag dran, damit das Programm weiss, mit welchen Komponenten es umgeht.

```
1 @FXML
   private TextField subjectField;
```

```
1 <TextField fx:id="subjectField" prefHeight="25.0" prefWidth="
   107.0" promptText="Subject"><TODO</TextField>
```

Da in beiden Dateien subjectField gleich heisst, kann man nun problemlos im Java Code auf das gewünschte Feld zugreifen. Man kann auch definieren, welche Methode aufgerufen werden soll, wenn man bsp einen Button drückt. Für addButtonPressed() und rmButtonPressed() haben wir das gemacht.

```
1 onAction="#addButtonPressed"
```

Im tag des AddButtons (in der fxml Datei) nennen wir die Methode die aufgerufen werden soll addButtonPressed". Damit der Java-Compiler merkt, dass er eine FXML Komponente suchen muss, fügen wir vor der Methode noch ein @FXML tag hinzu.

```
1 @FXML
   public void addButtonPressed() {
       ...
   }
```

Die addButtonPressed() Methode fügt Reminders in die Tabelle hinzu, solange der Input legal ist. Das heisst die Felder dürfen nicht leer sein (getValue != null). Wenn der Button gedrückt wird, werden die Felder wieder leer gemacht.

Die rmButtonPressed() Methode löscht Reminders die man mit der Maus selektiert hat.

```
1 ReminderList reminderSelected;
   reminderSelected = new ReminderList(reminderTable.
       getSelectionModel().getSelectedItem());
   model.removeReminders(reminderSelected);
```

Wir kreieren eine Liste und fügen alle selektierten Reminders hinzu, und löschen diese dann aus der Tabelle.

Die Methode Initialize haben wir reingenommen, da JavaFX Komponenten erst nach dem Ausführen des Konstruktors erstellt werden. Sie dient als Konstruktor.

```
1 BooleanBinding addBinding = subjectField.textProperty().  
    isEmpty().and(datePickerField.valueProperty().isNotNull()  
    ());  
    addButton.disableProperty().bind(addBinding.not());
```

Das Binding oben deaktiviert den addButton wenn der Input nicht valid ist.

```
1 reminderTable.getSelectionModel().setSelectionMode(  
    SelectionMode.MULTIPLE  
    );
```

Dies erlaubt dem User mehrere Items in der Tabelle zu markieren.

Um die Initialisierung des Models mussten wir noch ein try/ catch Block hinzufügen, da das Model Exceptions werfen kann.

```
1 reminderTable.setItems(model.getReminders());
```

Fügt beim Starten des Programms Reminders in die Tabelle, die in der DB gespeichert wurden.

4.2 Class: *DateTimePicker*

Die *DateTimePicker* Klasse haben nicht wir geschrieben. Wir haben diese Klasse auf StackOverflow gefunden. Da JavaFX nur einen *DatePicker* zur Verfügung stellt, mussten wir diesen benutzen. Es handelt mit *LocalDateTime* Objekten, die auch die Uhrzeit mitgeben. Der *DateTimePicker* gibt zwei sehr praktische Felder zur Verfügung: Das *dateTimeValue*, das das Datum inklusiver Zeit enthält, und das *valueProperty*, welches nur das Datum enthält. Im Konstruktor wird ein Listener erstellt, welcher das *dateTimeValue* überschreibt, wenn der User des GUI etwas ändert. In der *DateTimePicker* Klasse wurde noch eine zweite Klasse hinzugefügt, der *InternalConverter*. Dieser holt den String aus dem *DateTimePicker* und parsed es zu einem *LocalDateTime* Objekt.

5 Model

5.1 Class: *Model*

Die *Model* Klasse ist für die Daten Zuständig. Sie hat eine Liste mit allen Remindern als Klassenvariable. Die *Model* Klasse ladet die gespeicherten Daten über den *DataBaseAdapter*. Dies macht sie im Konstruktor. Die Methode *addReminder()*, *getReminders()*, *removeReminder()* und *removeReminders()* sind relativ selbsterklärend.

Die *bindData()* Methode ist ein bisschen interessanter. Da die *reminders* Liste eine *ObservableList* ist, kann man mit ihr ein Binding erstellen. Wir haben es so aufgebaut,

dass sobald sich irgendetwas in der Liste ändert, die Methode `adapter.save(reminders)` aufgerufen wird. So gehen die Reminders nicht verloren.

```
1 reminders.addListener(Poller.getInstance()::onChanged);
```

Dieses Statement macht zwei Sachen. `Poller.getInstance` gibt einen Poller zurück. Das `onChanged` macht, dass wenn sich etwas in der Reminder Liste ändert, der Poller notifiziert wird.

Die `redo()` und `undo()` Methoden kann man die Letzte Aktion in der Liste rückgängig machen. Das GUI selbst hat diese Funktionalität noch nicht, aber die Methoden machen auch wirklich das, was sie sollen.

5.2 Interface: DataBaseAdapter

Das DataBaseAdapter Interface stellt 2 Methoden zur Verfügung, die `save()` und die `load()` Methode. Save verlangt als Parameter die Liste von Reminders die Abgespeichert werden sollen.

5.3 Class: BinaryDBAdapter

Die BinaryDBAdapter Klasse implementiert das DataBaseAdapter Interface und Serialisiert die Binäre Datenbank unter dem Namen `reminders.ser`. Die `load()` Methode schaut zuerst ob die DB überhaupt existiert. Wenn nicht, wird eine neue erstellt. Dies machen wir mittels einem `ObjectInputStream`. Das Object das gespeichert wird (die Liste der Reminders), casten wir zu einer `ArrayList`, und diese `ArrayList` wird dann in eine `ReminderList` gefüllt. Die `save()` Methode wird immer aufgerufen, sobald sich irgendetwas in der Tabelle vom GUI ändert. Das Abspeichern machen wir mittels einem `ObjectOutputStream`.

5.4 Reminder

Der Reminder ist das Herzstück unseres Programms. Alles was in der Tabelle ist, ist ein Reminder. Ursprünglich waren die Klassenvariablen `SimpleStringProperties`, da die Tabelle solche verlangt. Das Problem war aber, dass sich diese nicht Serialisieren lassen. Also haben wir die Klasse ein wenig anders strukturiert. Anstatt `SimpleStringProperties` direkt in der Klasse zu speichern, werden diese einfach von den Methoden zurückgegeben. Das heisst anstatt wie folgt:

```
1 private SimpleStringProperty subject = new SimpleStringProperty
   ();

   public SimpleStringProperty getSubjectProperty() {
       return subject;
5  }
```

Machen wir es so:

```
1 private String subject = "";  
  
public SimpleStringProperty getSubjectProperty() {  
    return new SimpleStringProperty(subject);  
5 }
```

So hat man keine Probleme mehr mit dem Serialisieren.

5.5 ReminderList

Die *ReminderList* ist aus zwei Gründen entstanden. Wir hatten sehr häufig eine Beobachtbare (Observable) Liste von Reminden als inputparameter in Funktionen zu definieren. Dies verschlechtert die Leserlichkeit, welche bei Code sehr wichtig ist. Durch das definieren einer eigenen Klasse, kann man nun das besser lesbare *ReminderList* schreiben.

Ausserdem bietet uns die *ReminderList* auch die Möglichkeit den Zustand der *ReminderList* in einer History zu speichern, wenn man diese ändert. Somit kann man Änderungen an der liste später rückgängig machen. Durch das Hinzufügen einer *undone*-History, kann man nun auf der Zeitachse in beide Richtungen navigieren.

Leider spielte das *undo* / *redo* nicht gut mit JavaFx zusammen, so dass Änderungen gar nicht im GUI übernommen werden. Die entsprechenden Buttons im GUI haben wir deshalb wieder entfernt. Die *Undo/Redo* Funktionalität könnte aber in einem Nachfolgeprojekt Interessant werden. Deshalb haben wir diese nicht gelöscht.

5.6 Specific Reminder

Dies sind eine spezialisierung der Superklasse *Reminder*. Der einzige Unterschied besteht darin, dass “*SpecificReminder*” noch Tags speichern. Diese Tags können dann mit den Filtern gefiltert werden.

5.7 Poller

Der Poller folgt dem Singleton Pattern. Er läuft in einem eigenen Thread und ruft regelmässig (eingestellt ist jede Sekunde) den *NotificationHandler* auf. Der Poller kann als Observer auf die *ReminderList* “aufpassen” somit kann er jede Änderung welche vorgenommen wird sofort verarbeiten, und gegebenenfalls via *NotificationHandler* eine passende Notification absenden. Dies wird vor allem dann wichtig, wenn die Polling Zeit auf einen sehr grossen Wert gesetzt wird.

5.8 ConfigReader

Der *ConfigReader* müsste eigentlich von einer Config Datei lesen, die der User Anpassen kann. Dazu hat uns leider die Zeit nicht mehr gereicht. Im *ConfigReader* kann man einstellen ob man

- JavaFX Notifications überhaupt erhalten will,

- Notifications auf der Konsole haben will,
- ein Dark Mode haben will,
- vergangene Reminders sehen will
- Reminder sehen will, die in der nächsten Sekunde eintreffen
- Reminder sehen will, die im nächsten Monat eintreffen

5.9 Konfiguration

Die Konfigurationsmöglichkeiten sind im ConfigReader File zentral gelöst. Damit wird es später auch leicht möglich, über diese Klasse ein Konfigurationsfile einzulesen, und die entsprechenden Komponenten zu konfigurieren. Solange dies nicht über ein Externes File eingelesen werden, wird die Konfiguration direkt in den Boolean Werten gesetzt.

Bei den NotificationTypes muss man ein Objekt der gewünschten Notification eingeben, um einen neuen NotificationType zu konfigurieren. Dies hat den Vorteil einer Typesicherheit. Ausserdem kann somit der Reminder im Reminder.doNotify() einfach über die verschiedenen Notifications ittereieren, welcher er benutzen soll.

6 Filtering

6.1 Architektur

Wir wollten nach dem Objektorientierten Paradigma den einzelnen Klassen möglichst wenig wissen über die Internas von anderen Klassen zumuten. Somit sollte der Reminder selber prüfen ob er eine bestimmte Bedingung erfüllt und eine Notification absenden soll, anstatt seine Innereine dem NotificationHandler zu offenbaren.

Es drängte sich also auf dem Reminder eine Funtkiont zum testen zu übergeben. Ähnlich wie im Comand Pattern haben wir dies gelöst, indem wir Ein Objekt um die Funktion gewrappt haben. Anstatt execute() haben wir die funktion aber isTrue genannt, was ein gut lesbaren Quellcode mit den Tests erzeugt. Wie zum Beispiel der Criteriatester IsThisYear welcher einen Remeinder darauf testet, ojb er in diesem Jahr ist und die Antwort als boolean zurück gibt.

```
1 boolean isThisYear = IsThisYear.isTrue(reminder);
```

6.2 Higher Order Functions

Die Marketingabteilung von Oracle behauptet gerne Java sei auch Funktional. Higer Order Functions werden aber nicht wirklich unterstützt. https://en.wikipedia.org/wiki/Higher-order_function Java unterstützt leider keine ehcten Higher order functions. Man kann also nicht ohne weiteres eine Funktion als Inputparameter übergeben. Mittels Lambdas ist es lediglich möglich, eine Funktion ausführen zu lassen und den Rückgabewert als Inputparameter weiter zu verwenden. Dies erlaubt eine kompaktere

Notation. Dies reicht uns aber nicht, da wir den Remindern eine Funktion übergeben möchten, mit welcher jeder Reminder selber testet ob er eine Notification absenden soll.

6.3 echte Higher Order Functions in Java

Um dies zu erreichen haben wir eine Form von Higher Order Functions mit Objekten nachgebaut. Ein CriteriaTester ist ein Objekt, welches als Wrapper für eine Funktion dient. Anstelle dieser Funktion übergibt man nun diesen FunktionsWrapper als Inputparameter. Somit konnten wir Funktionen als Inputparameter mittels Objektorientierten Prinzipien nachbauen. Man muss nun für jede Funktion ein Objekt erstellen, welches das Interface CriteriaTester implementiert. Und die Filterfunktion isTrue implementieren. Funktionen als Rückgabewert kann man so aber noch nicht wirklich nachbauen. Für uns war das aber nicht nötig.

Dank den oben erwähnten Lambdas kann man dies auch elegant auf der Fly erledigen. Da es aber vorkommen kann dass man einen Filter mehrmals benutzt, haben wir uns entschieden die Filter jeweils als eigene Klassen zu implementieren.

6.4 Code Beispiel

Listing 1: on the fly criteria

```

1  Reminder reminder;
   Collection<CriteriaTester> criteria = new ArrayList<>();
   criteria.add(new IsPassed());
   //      example how CriteriaTester can be written on the
       fly
5  //put this to documentation
   criteria.add(
       reminder -> (!reminder.getTags().contains("
           hidden"))
   );
   //This lets the Reminder send a notification if the
       Reminder meets the criterias
10  //The first criteria it must pass is the IsPassed()
   reminder.notifyIf(criteria);

```

6.5 Implementierungen von Filtern

Wir haben etliche Filter implementiert, die meisten Filter ähneln sich stark. Die Dokumentation ist dementsprechend ähnlich. Für Klassen mit einer so grossen Ähnlichkeit wäre eine Programmierung mit Makros, wie sie beispielsweise C oder Lisp bietet eventuell angenehm. Da etliche Kriterien temporale Gegebenheiten testen, benutzen wir häufig LocalDateTime Objekte mit ihren Methoden. Wie bereits beschrieben steht vor allem die Criteria.isTrue() Methode im Vordergrund. Dementsprechend beleuchten wir hier auch nur diese Methode.

6.5.1 hasTag

Dieser Filter prüft, ob ein Reminder ein bestimmten Tag hat. Somit kann man zum Beispiel Reminders verstecken, wie im Beispiel weiter oben gezeigt, oder man kann Filter auch in andere Kategorien anordnen.

Das Filtern auf den Tag geschieht auf einer einzelnen Zeile.

```
1      public boolean isTrue(Reminder reminder) {  
        return reminder.getTags().contains(tag);  
      }
```

6.5.2 IsInNextMin

Dieser Filter testet, ob ein Reminder innerhalb der nächsten x Minuten stattfindet. Der default Constructor setzt x auf 1, so dass ohne nähere Spezifikation darauf getestet wird, ob ein Reminder in der nächsten Minute stattfindet. Dies kommt auch nahe an den natürlichen Sprachgebrauch, was den Code besser lesbar macht. Zuerst wird geprüft, ob der Reminder in der Zukunft liegt. Dann wird getestet, ob der Reminder innerhalb der nächsten nextMinutes stattfindet.

```
1      public boolean isTrue(Reminder reminder) {  
        return reminder.getDate().isAfter(LocalDate.now())  
            && reminder.getDate().isBefore(LocalDate.  
                now().plusMinutes(nextMinutes));  
      }
```

6.5.3 IsInNextSecond

Dieser Filter testet, ob ein Reminder innerhalb der nächsten x Sekunden stattfindet. Der default Constructor setzt x auf 1, so dass ohne nähere Spezifikation darauf getestet wird, ob ein Reminder in der nächsten Minute stattfindet. Dies kommt auch nahe an den natürlichen Sprachgebrauch, was den Code besser lesbar macht. Zuerst wird geprüft, ob der Reminder in der Zukunft liegt. Dann wird getestet, ob der Reminder innerhalb der nächsten nextSeconds stattfindet.

7 Notification

7.1 Interface: Notification

Das NotificationInterface gibt zwei Methoden vor:

```
1      void setReminder(Reminder reminder);
```

setReminder übergibt den Reminder, für den man eine Notification erstellen will.

```
1      void send();
```

send() wird vom Reminder aufgerufen, und zeigt dem User die Notification. Es gibt verschiedene Arten von Notifications. JavaFxNotification ist das Popup das den Reminder aufzeigt, der gerade aktuell ist. MultireminderNotification zeigt alle schon vergangenen Reminders, und ConsoleReminder zeigt auf der Konsole (Shell) einen Reminder. All diese Notification-Klassen implementieren Notification.

7.2 Class: ConsoleNotification

ConsoleNotification hat einen leeren Konstruktor aus mehreren Gründen. Wir haben ihn ähnlich wie den JavaFxNotification aufgebaut, und JavaFX verlangt einen leeren Konstruktor, also haben wir ihn hier beibehalten. Auch haben wir einen Konstruktor mit einem Reminder im Parameter gemacht, also war der DefaultKonstruktor überschrieben. Der ConfigReader benutzt diese Klasse auch (inklusive dem leeren Konstruktor) und benötigt keinen direkten Reminder, also haben wir ihn so stehen lassen.

Der Konstruktor mit dem Reminder im Parameter und die setReminderMethode machen eigentlich genau das gleiche. setReminders wird noch vom Notification Interface verlangt. Die send() Methode wird vom Reminder aufgerufen wenn die Zeit soweit ist. Sie druckt einfach die toString() Methode des Reminders auf die Konsole.

7.3 Class: JavaFxNotification

Die JavaFxNotification Klasse ist identisch mit der ConsoleNotification, nur die send() Methode ist anders. Die send() Methode enthält die statische Methode:

```
1 Platform.runLater( () -> { ... } );
```

In den geschweiften Klammern wird ein Popup Fenster erstellt. Da wir mit Threads arbeiten, und diese Popups verspätet aufgerufen werden, müssen wir das GUI in das Platform.runlater einpacken. Die JavaDoc vom runlater() sagt 'Run the specified Runnable on the JavaFX Application Thread at some unspecified time in the future [...] und das ist genau was wir brauchen. Lässt man es weg, bekommt man dutzende von Exceptions.

```
1 {
  Stage stage = new Stage();
    label = new Label("Hello: " + reminder.toString());
    Button okButton = new Button("Ok");
5    okButton.setOnAction(e -> {
      stage.close();
    });
    VBox pane = new VBox(10, label, okButton);
    pane.setAlignment(Pos.CENTER);
10    pane.setPadding(new Insets(10));
    Scene scene = new Scene(pane);
    stage.setTitle("Reminder");
    stage.setScene(scene);
```

```

15     stage.setResizable(false);
        stage.show();
    }

```

Der Code in den geschweiften Klammern macht ein simples JavaFX Fenster das den Reminder aufzeigt. Das Label wird mit der `reminder.toString()` Methode überschrieben. Dem `okButton` schliesst das Fenster wenn man ihn drückt. Die Komponenten `Button` und `Label` tun wir in ein `VBox` Behälter und machen ihn noch ein bisschen schöner mit `setAlignment()` und `setPadding()`.

7.4 Class: MultiReminderNotification

`MultiReminderNotification` benutzen wir um alle schon vergangenen Reminder in einem einzigen Fenster darzustellen. Es ist aber auch möglich, eine andere aggregation von Remindern mit ihr darzustellen. Der Aufbau der Klasse ist genau gleich wie in `JavaFxNotification` und `ConsoleNotification`, nur dass anstatt einem Reminder hat er eine Liste von Reminder.

```

1  private Collection<Reminder> reminders;

    String remindersText = "";
        int i = 0;
5      for (Reminder r : reminders) {
            remindersText += "Passed Event No " +
                ++i + ":\n";
            remindersText += r.toString() + "\n";
            System.out.print("added" + r.toString()
                );
        }

```

Hier Iterieren wir durch alle Reminders in der Tabelle und fügen sie zum Label hinzu. Die Reminders sind in der `reminders` Liste. Damit wir nicht alle Reminders in diesem Popup haben, sonder nur die die bereits vergangen sind oder schon sehr bald erscheinen, werden diese im `NotificationHandler` noch gefiltert. Die Methoden dazu wären folgende:

```

1  criteria.add(new IsPassed());
    criteria.add(new IsThisYear());

```

7.5 Class: NotificationHandler

Der `NotificationHandler` handelt die Notifications. Dazu werden die einzelnen NotificationTypen mit den passenden `CriteriaTesters` konfiguriert. Im Konstruktor wird dem `NotificationHandler` eine `ReminderList` übergeben. Für diese Reminders werden beim aufruf der `handel()` methode die Notifications verwaltet.

in der `handle()` Methode wird über die `ReminderList` itteriert. für jeden NotificationTyp werden nun die passenden `CriteriaTester` angegeben. Wir betrachten nun lediglich den

NotificationTyp, welcher sämtliche Reminders aufpoppen lässt, welche diesen Monat sind.

Listing 2: NotificationHandler.handle

```

1  Collection<CriteriaTester> importantStuffThisMonth = Arrays.
    asList(new IsThisMonth());
    if (!notifiedReminders.contains(reminder)) {
        /**
         * this passes the criteriaTesters to the Reminder itself, and
         * lets the Reminder send the notification if
5         * the criteria are met.
         */
        boolean success = reminder.notifyIf(importantStuffThisMonth
        );
        if (success) notifiedReminders.add(reminder);
    }

```

Dann wird für jeden Reminder, welcher noch keine Notification vom entsprechenden Typ abgesendet hat ein `Reminder.notifyIf(CriteriaTester)` aufgerufen. Der Reminder testet selbstständig, ob das Kriterium zutrifft, und er der Notification den Befehl gibt eine Meldung azusende. Falls dies geschieht, meldet der Reminder ein success zurück. Der Handler nimmt ihn dann in die Liste der Reminder auf, welche bereits eine Notification abgesendet haben.

Kurz vor dem Datum eines Reminders, wird nochmals auf diesen Reminder hingewiesen. Der Code funktioniert sehr ähnlich. Es wird aber anstatt ein `CriteriaTester.IsThisMonth()` ein `CriteriaTester.IsNextSecond()` übergeben.

7.5.1 Methode showPastEvents

Es gibt noch ein dritten Typ von Notifications diese zeigt eine Aggregation der vergangenen Reminders. Sie werden in einer separaten Methode abgearbeitet. Der `showPastEvents()` Methode. Diese löst eine Notification aus, welche mehrere Reminders zusammen darstellt. Deshalb haben wir uns von dem Paradigma gelöst, dass nur der Reminder die Notification auslöst. Die Methode überprüft auf zwei Kriterien. Erstens ob der Reminder in der Vergangenheit angesiedelt ist und ob der Reminder in diesem Jahr war. Dies geschieht indem man über die `ReminderList` iteriert, und die passenden Reminders in eine separate Liste namens `passedReminders` speichert. Diese wird dann der `MultiReminderNotification` übergeben, damit diese die aggregierte Notification absenden kann.

Listing 3: NotificationHandler.showPastEvents

```

1  public void showPastEvents() {
        ArrayList<Reminder> reminderList = reminders.
            getSerializable();
        ArrayList<Reminder> passedReminders = new ArrayList<>()
        ;
    }

```



```
Collection<CriteriaTester> criteria = new ArrayList<>()
;
5  /**
   * the both criteria filter the Reminders for Reminders
   * , which are dated in th past and dated this year.
   */
criteria.add(new IsPassed());
criteria.add(new IsThisYear());
10
for (Reminder reminder : reminderList) {
    if (reminder.meetsCriteria(criteria))
        passedReminders.add(reminder);
}
15 // gets sure that a Notification is only sent, if it is
    not void.
if (passedReminders.size() != 0) {
    new MultiReminderNotification(passedReminders).send
        ();
}
}
```

Um sicherzustellen, dass die `showPastEvents` nur einmal dargestellt werden, muss der Poller sich merken, ob die Methode schon einmal aufgerufen wurde. Falls dies zutrifft, wird auf ein weiterer Aufruf verzichtet.

```
1 public boolean isTrue(Reminder reminder) {
    return reminder.getDate().isAfter(LocalDateTime.now())
        && reminder.getDate().isBefore(LocalDateTime.
            now().plusSeconds(nextSeconds));
}
```

`subsubsectionIsPassed` Diers Filter testet, ob ein Reminder in der Vergangenheit angesiedelt ist. Dazu wird die `LocalDateTime.isBefore` Methode benutzt um zu testen, ob der Reminder vor dem jetzigen Zeitpunkt stattfindet.

```
1 public boolean isTrue(Reminder reminder) {
    return reminder.getDate().isBefore(LocalDateTime.now())
        ;
}
```

7.6 *IsThisYear*

Dieser Filter testet ob ein Reminder diesen Jahr stattfindet. Wir vergleichen dabei das Jahr des Reminders mit dem aktuellen Jahr.

```
1 public boolean isTrue(Reminder reminder) {
```

```
        return reminder.getDate().getYear() == LocalDateTime.  
            now().getYear();  
    }
```

7.7 IsThisMonth

Dieser Filter testet ob ein Reminder diesen Monat stattfindet. Dazu testen wir ob der Reminder im selben Jahr und in demselben Monat stattfindet.

```
1      public boolean isTrue(Reminder reminder) {  
        LocalDateTime today = LocalDateTime.now();  
        return reminder.getDate().getYear() == today.getYear()  
            && reminder.getDate().getMonth() == today.  
                getMonth();  
5    }
```

7.8 IsToday

Dieser Filter testet ob ein Reminder an diesem Tag stattfindet. Dazu testen wir ob der Reminder im selben Jahr und in demselben Monat und am selben Tag stattfindet.

```
1      public boolean isTrue(Reminder reminder) {  
        LocalDateTime today = LocalDateTime.now();  
        return reminder.getDate().getYear() == today.getYear()  
            && reminder.getDate().getMonth() == today.  
                getMonth();  
5    }
```

8 Resources

8.1 Resources

Die Ressourcen die wir gebraucht haben waren:

- Java
- JavaFX
- IntelliJ
- Atom
- Latex
- TornadoFX-Controls Plugin
- Git
- Github

9 Tests

10 Tools

Um die Versionierung der Dokumentation automatisch generieren zu lassen, haben wir LaTeX so mit Scripts erweitert, so dass die git Head Versionsnummer direkt ins Dokument eingefügt wird. Somit bleibt diese Information auch auf einem Ausdruck akkurat. Gegenüber einer manuellen inkrementierung der Version, hat die automatisierung den Vorteil, dass sie auch im Stress nicht vergessen wird. Das Ergebniss sieht man am Ende des Dokuments.

LaTeX kann Code ausführen. Wir haben den Code in Shellscrip te ausgelagert, und lassen den compiler diese aufrufen. Damit dies funktioniert, muss dies aktiviert werden.

Im LaTeXfile steht nun folgender Code. Zuerst wird die Versionsinformation in eine Datei geschrieben, welche anschliessend in die Dokumentation eingebunden wird. Am schluss wird die Datei zurückgesetzt, so dass sie eine Fehlermeldung im Dokument erzeugt, falls die Ausführung von Shellscrip ten im LaTeXcompiler ausgeschaltet ist. Die automatisch generierte Datei wird nach der Generierung in das Dokument eingebunden.

```

1 \noindent
  Automatische Versionierung:
  \immediate\write18{../script/versionInfo.sh}
  \input{../script/version}
5 \immediate\write18{../script/cleanup.sh}

```

Wir schreiben den output in eine version.tex Datei. Auf Zeile 9 lassen wir git HEAD nummer in die Datei speichern, welche beim pushen jeweils inkrementiert wird.

```

1 #!/bin/sh
  OUTPUT="../script/version.tex"

  echo "Last compiled: ">$OUTPUT
5  date >> $OUTPUT

  echo "\n">>$OUTPUT

  echo "Git HEAD Version: ">> $OUTPUT
10 git rev-list --count --first-parent HEAD >>$OUTPUT

```

Dann wird ein cleanup durchgeführt, dabei wird die Output Datei mit einer Fehlermeldung versehen, so dass der User bemerkt, falls die Automatische Versionierung fehlschlägt.

```

1 #!/bin/sh
  OUTPUT="../script/version.tex"
  echo "Fetching version information failed. Please enable shell-
    escape in your \LaTeX \~ compiler.">$OUTPUT

```

11 Versionskontrolle

Manuelle Version: 1.0.0

Automatische Versionierung: Last compiled: Fri 16 Jun 14:52:53 CEST 2017

Git HEAD Version: 203