Object-Oriented Programming - a.y. 2013/2014
B.Sc. Degree in Computer Science & Engineering
University of Bologna - Campus of Cesena
April 29th, 2014

Documentation about the engineering of

FiDO: an Android™ app to manage your pets

a paper by Nicola Giancecchi and Michele Sapignoli

"A dog teaches a boy fidelity, perseverance, and to turn around three times before lying down."

— *Robert Benchley*

CONCEPT 4

01 The idea 4

02 Problem analysis 4

ENGINEERING 6

03 Architectural design 6

03.01 Model 6

03.02 View 8

03.02.01 Activities 8

03.02.02 Adapters 9

03.03 Controller 11

04 User Interface design 12

05 Packages organization 12

06 Work division 13

07 Detailed engineering 14

07.01 Nicola Giancecchi's work 14

07.01.01 App's fundamentals 14

07.01.02 Measures 14

07.01.03 Diet 15

07.01.04 Patterns 16

07.02 Michele Sapignoli's work *18*

07.02.01 Reminder 18

07.02.02 General Info 20

07.02.03 Notification system 21

07.01.04 Patterns 22

07.03 The common part *22*

07.03.01 Main activity 22

07.03.02 Stats & Costs 22

DEPLOYMENT **23**

08 Testing 23

09 Final considerations 23

09.01 Workflow *24*

09.02 Interactions & criticalities *24*

10 Next steps 25

11 Bibliography & credits 25

12 Group contact informations 25

CONCEPT

01 The idea

"Ehi Nico, cosa ne dici se facessimo una bella app per gestire i nostri animali domestici?" (Ehi Nico, what do you think about develoving an app for our pet friends?).

This is how FiDO was born. A simple idea. A smart idea. An idea that has become reality.

Since we were children, our love for pets has always been strong. We grew up surrounded by the companionship and the warmth of our four legged friends: Max and Stitch. Their love has changed us, and inspired us as well to make a colorful and essential application, able to manage all the needs and treatments that a pet needs from its owner.

Since the beginning, we wanted to create a powerful, useful but simple mean to deal with all our beloved friends. We knew that overcoming this challenge was not a joke, but we have done our best in order to offer to FiDO users an application that could assist them in their ordinary life with their pets.

We actually had started working on this product weeks before we wrote the first line of code. In fact, FiDO was deeply studied and analized with the goal of making the user experience easy and comfortable, as much as possible.

02 Problem analysis

What features does an application have to have, in order to deal with pets?
This is the main question that occupied our minds for the first weeks.

After a deep analysis, we indentified some basic ones, so our application should have been capable of:

- Adding a single owner of the pets.
- Adding a several number of pets.
- Managing pets info.
- Deleting a pet.
- Adding different types of operations for pets.
- Editing and deleting pets operations.
- Showing pets statistics.

Furthermore, we focused on the kind of tasks that a pet could have and we chose some important features of the app:

- **Reminder section**: a list of upcoming and past events related to the pet's life, with the possibility to add and edit events of different types.
- **Diet section**: a side completely devoted to the pet's feeding, realized through the making of a weekly diet.
- **Measure section**: a part of the app used to add and manage daily measurements.
- **Stats section**: a tool to track pet growth, showing pet's measurements and owner's costs.
- **Notifications**: a system that notifies the user for events and pet's meals.
- **Data loading system**: realized on .dat file.

The first analysis was also intended to choose the platform of our application, desktop or mobile, in order to design the corresponding GUI. The team considered the advantages and disadvantages of both of them and, in the end, preferred the mobile, mainly because of a wider and more frequent possible usage of the app.

For this reasons we chose to develop FiDO for Android devices.

ENGINEERING

03 Architectural design

The app basically uses the **MVC (Model-View-Controller)** development pattern. MVC allows to separate the roles for data definition, manipulation and presentation. In Android, the view component of the pattern is composed by two fundamental parts:

- **Activities**, that manages the viewing of data on the device and update it after inserting, editing or deleting data

- **Android XML Interface Files**, that specificate the static design of the view (like Visual Studio's Form files and Xcode's XIB files) and includes all the controls of the view - such as buttons, textviews, listviews, etc…

Other Java patterns are also used in the app, such as the **Template Method** - used to reduce significantly the duplication of code — and **Singleton** — used in app's `DataHelper` class to manage the I/O, it makes available a single reference of the saved file through the whole app. Inherently, the app makes use of the **Adapter** pattern by using it in the Adapters for the Android ListView UI controls (they are on the `com.oop1314.fido.gui.adapters` package). These patterns will be showed later in the *Detailed engineering* section.

03.01 Model

Description of the main aspects:

- **Pet**: class with all the animal data, which has getters and setters. In order to guarantee the persistence of the data, the class implements **Serializable**. It implements **IPickerHelper** as well.

- **Owner**: class with all the owner's data , which has getters and setters. It handles personal data, the selected **Pet** in the MainActivity and the default time for Reminder notification. It implements **Serializable.**

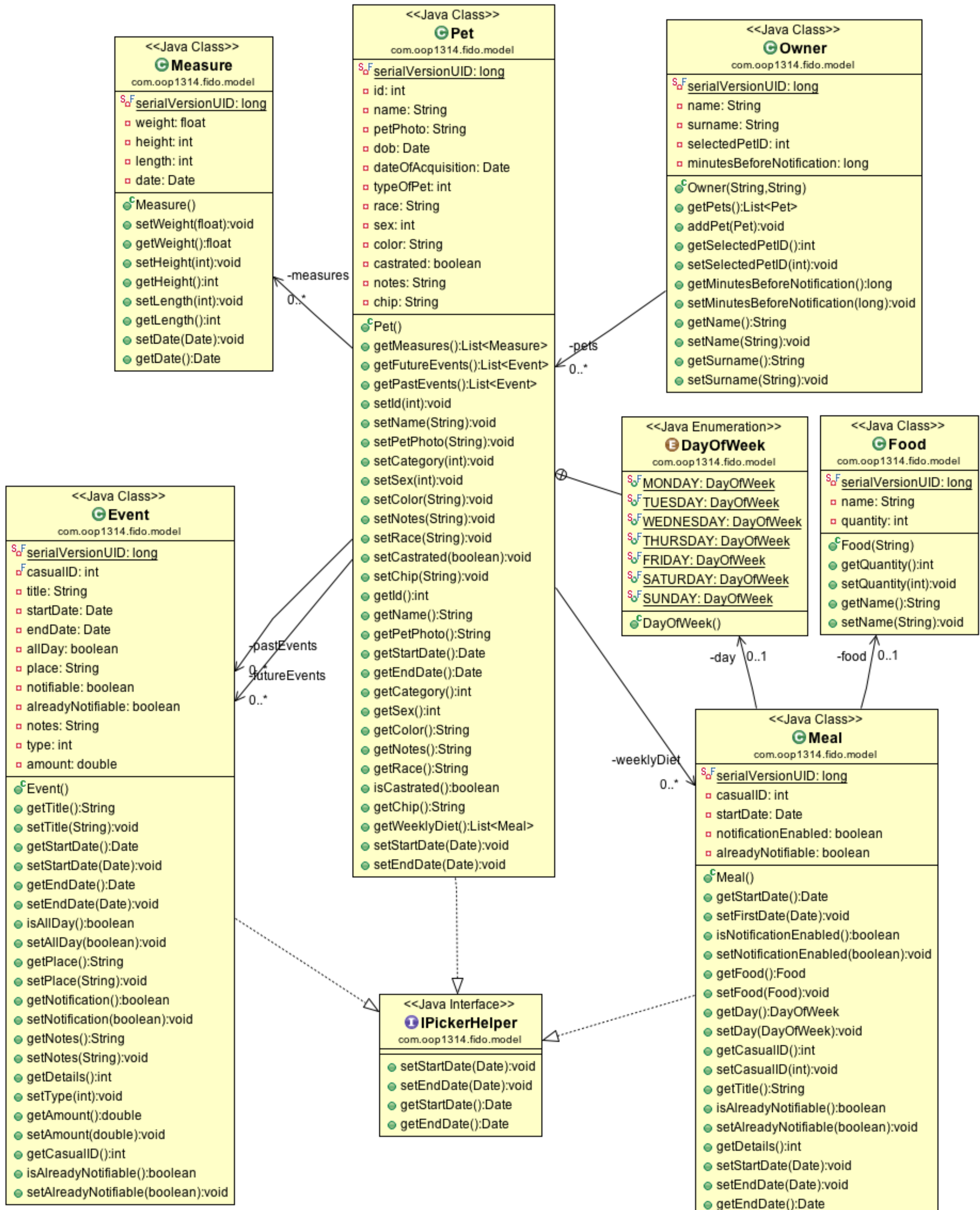
- **IPickerHelper:** interface used to simplify the handling of date and time pickers.

- **Event:** class that models an event, which has getters and setters. It implements **Serializable** , **IPickerHelper** and **INotification** (see p.19).

- **Meal**: class that models a meal, which has getters and setters. It includes a **Food** field. It implements **Serializable** , **IPickerHelper** and **INotification** (see p.19).

- **Food**: class that describes a food, with name and quantity, for the pet. It implements **Serializable.**

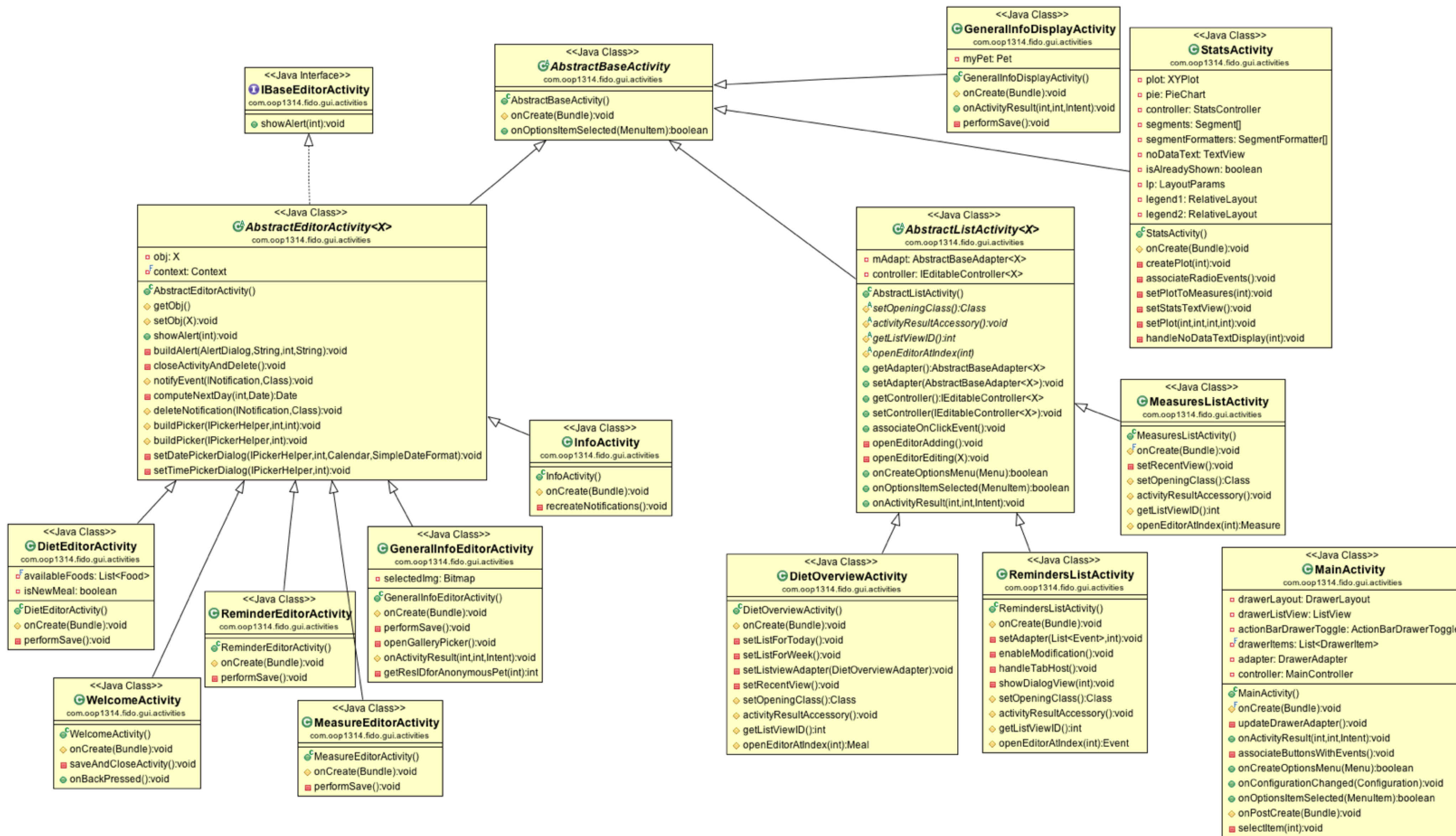
- **DayOfWeek:** enumeration for days.



03.02 View

03.02.01 Activities

UML diagram of the View Activities section.



- The **WelcomeActivity** is the activity the user sees at the first start of the app. It is used to collect name and surname of the owner. Once completed, a **GeneralInfoEditorActivity** is shown. It is made for building a **Pet** object, with all the info regarding the animal.

- Both WelcomeActivity and GeneralInfoEditorActivity extend **AbstractEditorActivity**, an abstract generic class that implements common methods for Editors. These methods are helpful in order to show alerts, build pickers or notify events. The class has a generic X field too. It can be an object from Pet, Owner, Event, Meal, Measure or Long class. Every class that extends AbstractEditorActivity has a different usage of the X field.

- The abstract class **AbstractBaseActivity** is the class at the top of the activity pyramid and it is needed for displaying the return button, at the left top of the activities.

- The abstract class **AbstractListActivity** is part of the template method for list activities (see p. 16). It is composed by common methods for list activities (DietOverviewActivity, MeasuresListActivity, RemindersListActivity). It has generic X fields for an Adapter and a Controller and corresponding methods for setting and getting them. Other methods are for opening/editing objects and handling results from editor activities.

- The **StatsActivity** shows plots and info about the growth and the costs related to the pet.

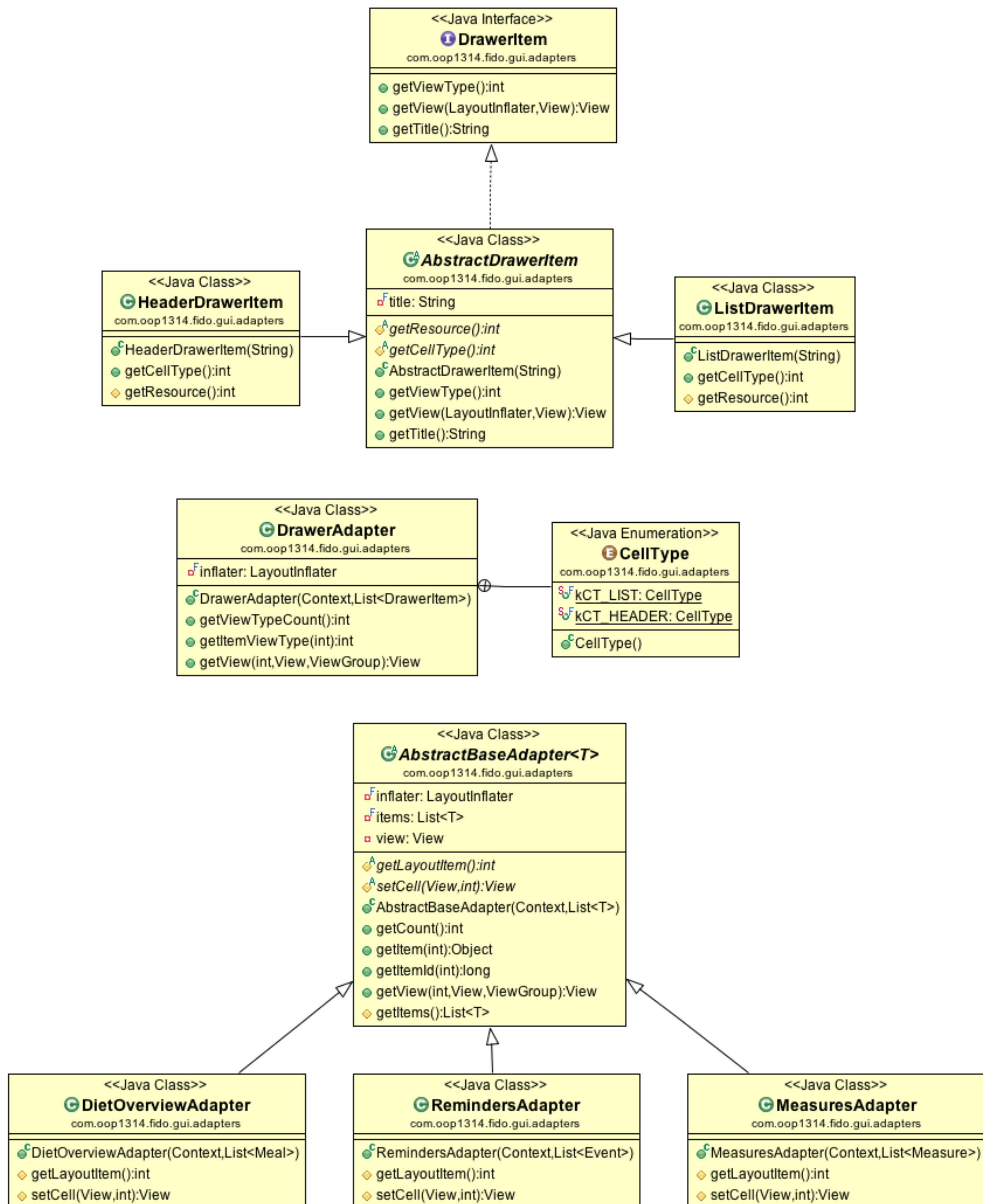
- The **MainActivity** is the primary view of the app. It is composed by a large clickable image and four buttons, connected to GeneralInfoDisplayActivity, RemindersListActivity, DietOverviewActivity, MeasuresListActivity and StatsActivity.

03.02.02 Adapters

Fido Adapters implements the fundamental methods for giving the user an easy-to-use interface.

- The abstract class **AbstractBaseAdapter** is part of a template method pattern. It includes common code for the list design in the activities. Its implementation differs in the three classes **DietOverviewAdapter**, **RemindersAdapter** and **MeasuresAdapter**. Each class creates a different style for its own list.

- The classes used to manage the lateral drawer are the following: **DrawerAdapter** is the adapter designed for the lateral navigation Drawer control. **DrawerItem** and **AbstractDrawerItem** are the interface and the abstract class that have common methods declarations and implementation. The class **ListDrawerItem** is designed for Pet's names and cells which are located in the lateral drawer. The **HeaderDrawerItem** class, instead, is formed by methods for the cell titles.


<<Java Interface>>
DrawerItem
com.oop1314.fido.gui.adapters
getViewType():int
getView(LayoutInflater,View):View
getTitle():String
<<Java Class>>
HeaderDrawerItem
com.oop1314.fido.gui.adapters
HeaderDrawerItem(String)
getCellType():int
getResource():int
<<Java Class>>
AbstractDrawerItem
com.oop1314.fido.gui.adapters
title: String
getResource():int
getCellType():int
AbstractDrawerItem(String)
getViewType():int
getView(LayoutInflater,View):View
getTitle():String
<<Java Class>>
ListDrawerItem
com.oop1314.fido.gui.adapters
ListDrawerItem(String)
getCellType():int
getResource():int
<<Java Class>>
DrawerAdapter
com.oop1314.fido.gui.adapters
inflater: LayoutInflater
DrawerAdapter(Context,List<DrawerItem>)
getViewTypeCount():int
getItemViewType(int):int
getView(int,View,ViewGroup):View
<<Java Enumeration>>
CellType
com.oop1314.fido.gui.adapters
kCT_LIST: CellType
kCT_HEADER: CellType
CellType()
<<Java Class>>
AbstractBaseAdapter<T>
com.oop1314.fido.gui.adapters
inflater: LayoutInflater
items: List<T>
view: View
getLayoutItem():int
setCell(View,int):View
AbstractBaseAdapter(Context,List<T>)
getCount():int
getItem(int):Object
getItemId(int):long
getView(int,View,ViewGroup):View
getItems():List<T>
<<Java Class>>
DietOverviewAdapter
com.oop1314.fido.gui.adapters
DietOverviewAdapter(Context,List<Meal>)
getLayoutItem():int
setCell(View,int):View
<<Java Class>>
RemindersAdapter
com.oop1314.fido.gui.adapters
RemindersAdapter(Context,List<Event>)
getLayoutItem():int
setCell(View,int):View
<<Java Class>>
MeasuresAdapter
com.oop1314.fido.gui.adapters
MeasuresAdapter(Context,List<Measure>)
getLayoutItem():int
setCell(View,int):View

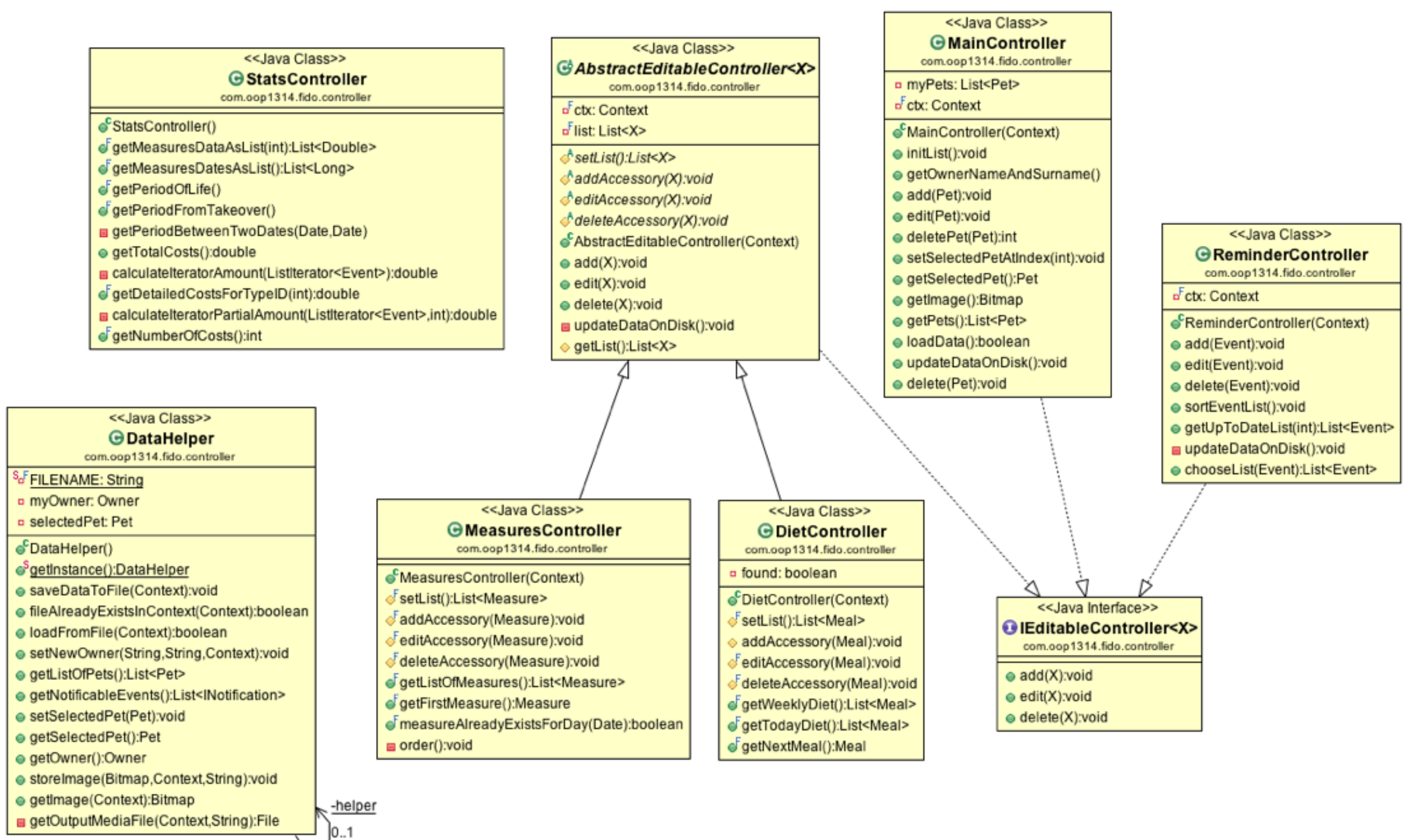
03.03 Controller

- As **MeasureController** and **DietController** have some common code, the abstract class **AbstractEditableController** is a factorization in a template method of the common methods for these classes. Its aim is to write once the code for adding, editing and deleting a generic X object (Measure or Meal) and for saving data on file. Each implementation has different controls and methods for its own needs. Being **ReminderController** a little bit different from the others, it shares only the **IEditableController** implementation. The ReminderController handles data differently from the other controllers, having two lists of events instead of one in Pet class.

- The **MainController** class does all the "dirty work" that MainActivity needs. It has a list of pets and methods for adding, editing, deleting and handle them.

- The **StatsController** class has methods to calculate the total amount of costs and the detailed amount for each Event type.

- The **DataHelper** class is the singletone, used to load and save data between the various views (see p.18).



04 User Interface design

A particular attention is focused on the app's user interface & experience design since the beginning of the project. Some graphical sketches were made before the project approval to have an idea of the app behavior.

UI elements and call to actions are clear, easy to understand and in line with the actual mobile design trends.

05 Packages organization

The app's source code is organized into these packages, under the main package **`com.oop1314.fido`**.

- `com.oop1314.fido.`**`model`**: contains all the model classes of the app.

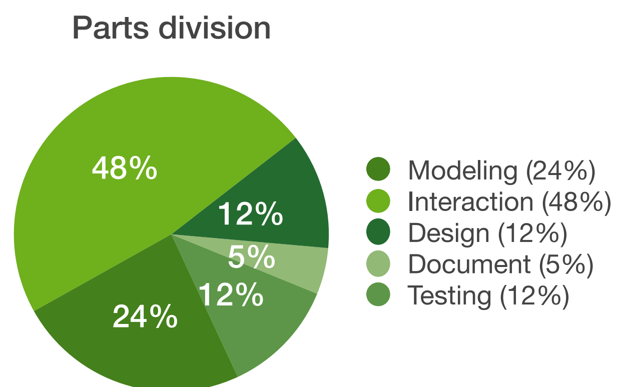
- `com.oop1314.fido.gui`: contains the classes about the view and it's organized into two subpackages:
    - `com.oop1314.fido.gui.activities`: contains all the activities of the app that manage the viewing of data.
    - `com.oop1314.fido.gui.adapters`: contains the view's adapters, that help the app to show data into Android ListView items.
- `com.oop1314.fido.controller`: contains the controller classes that intermediate the dialogue between model and view classes.
- `com.oop1314.fido.exceptions`: contains the classes used for throwing exceptions.
- `com.oop1314.fido.notifications`: contains the classes that manage local notifications.

Javadoc for this project is available into the `/docs` folder.

06 Work division

All the work (excluded the making of the idea and the first app concepts) was performed in about **200 hours**, divided equally between Giancecchi and Sapignoli. The common activities took about 20 hours (20%) for each resource. All the hours were tracked through the *Podio* system (www.podio.com).

The features were divided following this scheme:



Nicola Giancecchi:

Measure Tracking section, Diet section. Activities and Controllers for Measures and Diet. Exceptions. Adapters. Left Drawer. Photo handling in pet's info.

Michele Sapignoli:

General Info section, Reminder section. Activities and Controllers for General Info and Reminder. Pet addition/deletion. Notification system. Beta testing on real devices.

Common part:

Main Activity. Stats & Costs + Info section. Activities and Controller for Stats & Costs + Info. Constants.

07 Detailed engineering

07.01 Nicola Giancecchi's work

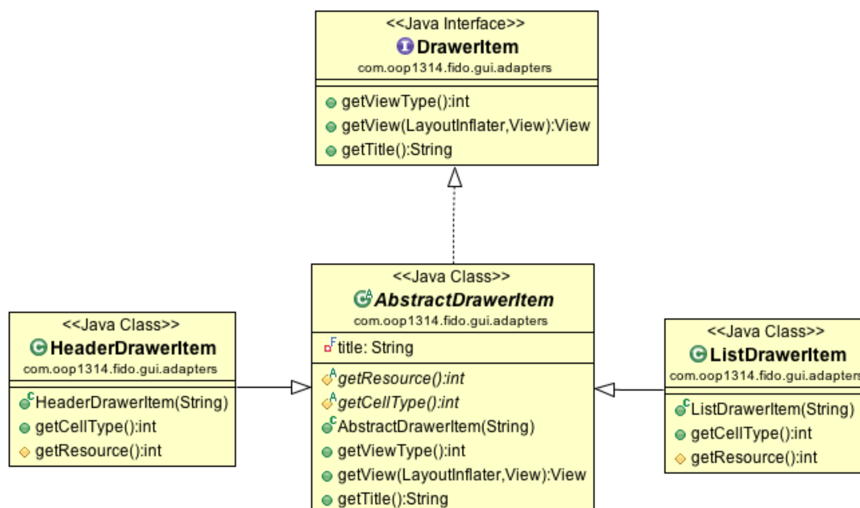
This section is under the responsibility of the resource Nicola Giancecchi.
Will be illustrated his share of work in the project.

My work was initially focused on the general structure and the visual fundamentals of the app. Then I worked on the building of Measures and Diet sections, that are very similar in their structure. Next, together with Michele we have built the Stats section and refactored the reused code into the Java main patterns.

07.01.01 App's fundamentals

In the days before the approval of the project we've sketched out the various views of the app and the general structure. In particular, it was decided to use the Navigation Drawer: an Android control used in many apps and in line with current trends (known on iOS for its first appearance in the Facebook app).
The main view has been designed to be understandable and has direct links to the major parts of the app.

In particular, the Navigation Drawer structure is under a template method:



07.01.02 Measures

The first section implemented by me is **Measures**. The main class for this section is Measure: it has three main fields that are the characteristic of this section — weight, height and length — and one Date field that can be considered as the primary field of the class: thanks to the relative Controller methods that I'll show next, there must be only one Measure object for each day. The Measures methods are only getters and setters.

These objects are showed into the **MeasuresListActivity** that can be called by tapping on the “Measures” button on the Main Activity. In this view you can simply see on the top of the view the last Measure entered. Immediately after are showed all the measures inserted by the user.

All the logic of this view is managed by the **MeasuresController** class, into the controllers package. It has methods for adding, editing and deleting a Measures that belongs to the **AbstractEditableController** template method pattern. It also incorporates methods for ordering the list, checking if a Measure already exists in a day and getting the first Measure to be shown in the top of MeasuresListActivity. With the same view, if you tap on a ListView item on the List activity, you can edit a measure that already exists. In this case a red “Delete this measure” button will be shown to give to the user the possibility of deleting a Measure.

By tapping the “+” button on the Navigation Bar, you can add a Measure through the **MeasureEditorActivity**. Into this view you can add or edit a measure by filling the three TextView fields and setting the date through the date picker. Since a measure in a day is unique, if you want to edit a Measure, the date picker will be disabled.

Tapping on “**Save**” button, the activity will build — or edit if it already exists — an object of Measure type that will be passed to the MeasuresListActivity as an Intent result. Here, with the AbstractListActivity template method, the object will be managed and saved onto disk.

07.01.03 Diet

The second section implemented is **Diet**. This is a little bit more difficult section than Measures that required some more hours of development.

The Diet associated class is called **Meal**. It has some fields: the ID as primary key, a DayOfWeek enumerator that represents the choosen day, a “startDate” (hour of serving), a Food object that contains the name of the food and the quantity, and two flags that affects Notifications, implemented by my colleague.

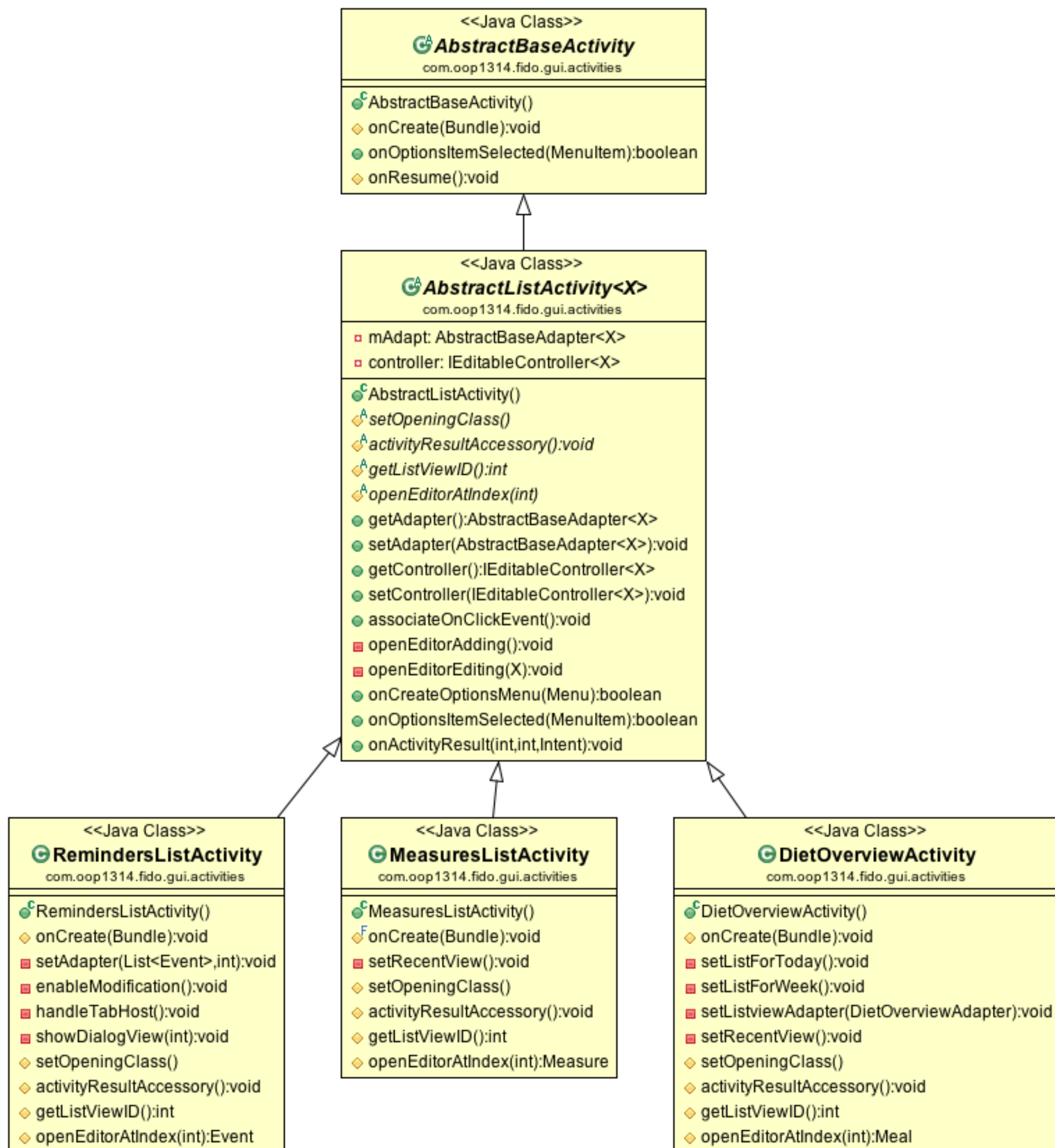
As Measures, Diet is accessible from the Main Activity. The view is also similar to Measures: it has a top view that shows the next meal to be served to the pet and a ListView. Through the switch that is inside the view, you can choose if you want to show into the ListView the diet for today or for the whole week.

By tapping the “+” button, you can add a new Meal through the **DietEditorActivity**. Here you can select the day by using a *NumberPicker* adapted with string values, the hour using a *TimePickerDialog*, the desired meal (Croquettes/Paste/Fodder/Targeted food/Snacks/ Supplements), the quantity in grams and choose if you want to receive a local notification.

Like measures, tapping on Save button will generate an object that will be managed by the DietOverviewActivity.

07.01.04 Patterns

Understanding the AbstractListActivity template method



It was necessary to implement the template method into classes of "Activities List" type that have some common methods in which only a few parameters changes.

The methods `openEditorAdding()` and `openEditorEditing(final X obj)` were identical on all List views, except for the Intent constructor that requests the Class of the view. I've solved this by calling the abstract method `setOpeningClass()`.

The method `associateOnClickEvent()` was identical on all List views, except for the logic of retrieving the data to be edited. I've solved this by calling the abstract method `openEditorAtIndex(int arg)` that returns the desired object at the passed index.

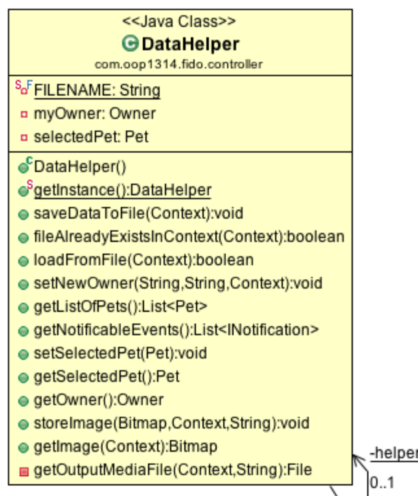
The method onActivityResult(final int requestCode, final int resultCode, final Intent data) was the same on all List views. I've put this on the abstract class and added an activityResultAccessory() method that is called if I would do some accessory operations — e.g. sorting — after adding/editing/deleting the object from the list.

Understanding the AbstractEditableController template method

It was necessary to implement the template method into the Editable controllers due to presence of duplicated code and for having a single `performSave()` method reference to the DataHelper class.

AbstractEditableController is the solution for easily build a controller for adding/editing/deleting objects in a List of type X.

The method `setList()` sets the list where editing operations should be applied. The logic for manipulating data into the list is delegated to the methods `addAccessory()`, `editAccessory()` and `deleteAccessory()`. Everytime one of these methods (add/edit/delete) are called, DataHelper will save the Owner object onto disk.



Understanding the DataHelper singleton

We decided to implement a singleton for the management of I/O because being the object to be manipulated only one in the whole app (the .dat file), the management of the references to the objects and their access becomes easier.

DataHelper provides methods for saving and reading the Owner serializable object and for suppling to the Controller classes the main Owner's fields — like the list of pets or the notifications. It also manages the storage and the reading of images saved on disk.

07.02 Michele Sapignoli's work

This section is under the responsibility of the resource Michele Sapignoli.
Will be illustrated his share of work in the project.

My first goal was to create a well-functioning Notification system and well-designed General Info and Reminder sections. I started working on this last section first, because I knew it could have been the hardest part to deal with.

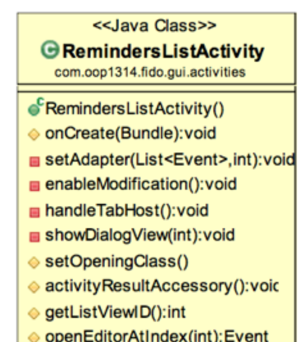
07.02.01 Reminder

The **Event** class is the base class of the whole Reminder section. It is composed by many fields, but the essential ones are the title and the date (beginning date and ending date).

Being the raw material of the section, the objects of this class are built, by tapping the "+" button, by the **ReminderEditorActivity**, which is a part of the GUI completely devoted to the construction of an Event.
This activity shows several TextFields to fill, two CheckBoxes used for Notifications and All-day-activity and two Buttons. It also allows to add Event costs.

Once pressed, the buttons show a *DatePickerDialog* and, immediately after, a *TimePickerDialog*. This action is perform through the usage of the *IPickerHelper* Interface, which is used by the abstract class **AbstractEditorActivity** (the one -among the others- extended by **ReminderEditorActivity** and **DietEditorActivity**) in order to show the *DatePickerDialog* and the *TimePickerDialog*.

This class incorporates common methods for the Editors (**DietEditorActivity, GeneralInfoEditorActivity, MeasureEditorActivity, ReminderEditorActivity**).
It is composed by a generic field X, which is the raw type of the object for the corresponding activity (Meal, Pet, Measure, Event), and methods such as *showAlert* or *notifyEvent*, which are used by the Editors.
The methods *setTimePickerDialog* and *setDatePickerDialog* take an **IPickerHelper** object in input: in this way they can work both for Pets and Events objects.



I made this decision not to duplicate the code for showing the Dialogs in the **GeneralInfoEditorActivity** (for Pets) and in the **ReminderEditorActivity** (for Events).
My colleague Nicola Giancecchi separated the two methods (before together) in order to use the *TimePickerDialog* for his **DietEditorActivity**.
Once an Event object is built, it is sent to the **RemindersListActivity**, thanks to the *performSave* method.

The part of the Activity Lists is implemented through the usage of the **Template Method**, described in my colleague's work.

What **RemindersListActivity** do, is being a display of two lists of events: upcoming and past.
The **ReminderController** plays a fundamental role for this part of the GUI: it adds, edits and deletes objects from the list as the other controllers but, furthermore, it sorts the future lists, putting events in order (*sortEventList*), and checks if they are up-to-date (*getUpToDateList*). If an upcoming Event is pressed, **ReminderEditorActivity** opens up with the possibility to edit the event. If a past Event is pressed, an alert comes out, showing notes and giving the possibility to add/edit the amount. Amounts will be displayed in the **StatsActivity**.

The look of the cell is described in the `reminders_listview_item.xml`, located in the `res/layout` folder.

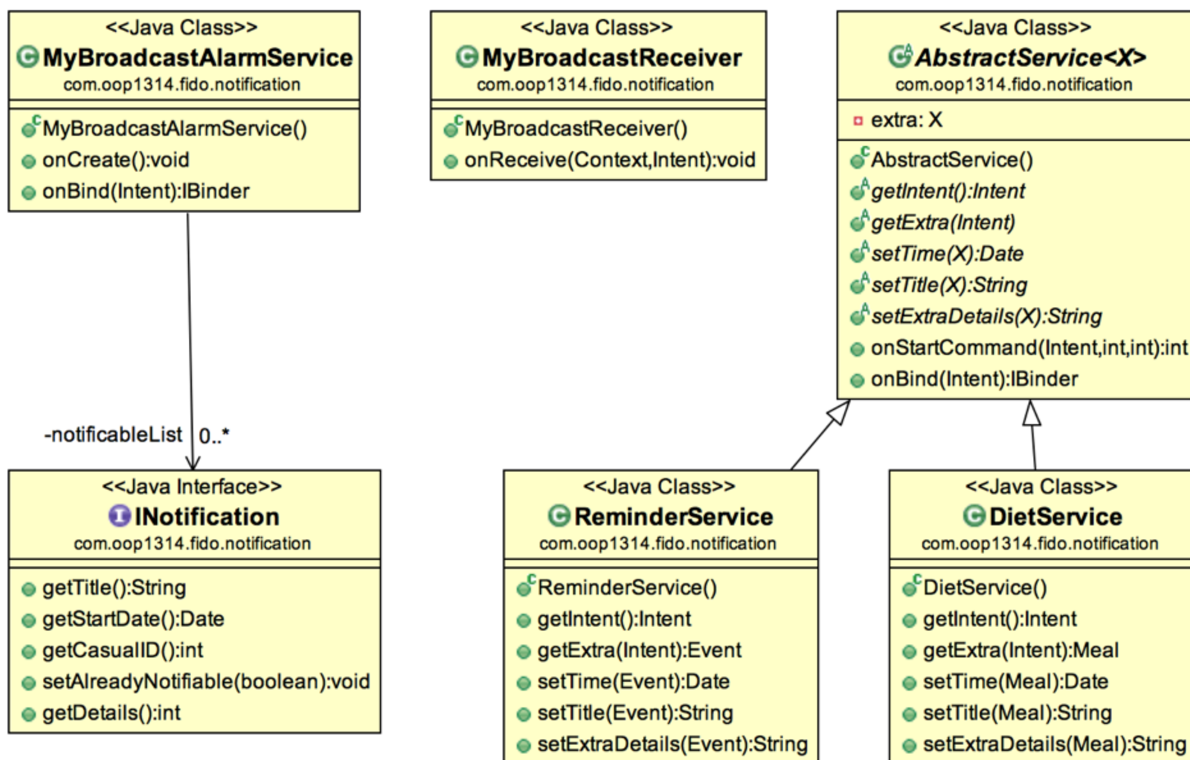
I found this section not so easy to develop. Linking the Date & Time Dialogs, dealing with up-to-date lists in the *TabHost* and passing data was not immediate. But after this large part of work, I could concentrate on the GeneralInfo section, with a useful welth of knowledge from the Reminder section.

07.02.02 General Info

The GeneralInfo section is composed by an Editor and a display Activity too.
The user can enter this section by tapping the large image in the **MainActivity**. The first screen the user sees is the **GeneralInfoDisplayActivity***:* a simple identity card of the pet. The entire layout is formed by TextViews displaying the name and the info about the animal.

By clicking the Edit button, the user opens the **GeneralInfoEditorActivity***.* In this piece of GUI, it is possible to change the details and the photo of the pet and to add notes. At the end of the page, there is a red button which gives the user the possibility to delete the current pet. The action will fail if the owner's current pet is the only one remained, in fact an Alert (made through the use of the method *showAlert* in the abstract class **AbstractEditorActivity**) will notify the impossibility to complete the action.
If the pet is deleted, the app will bring you to the **MainActivity**, otherwise, once the edit is complete, the modified Pet object is sent to the **GeneralInfoDisplayActivity** and here a confirm is required, in order to save the modified object.

The most difficult part of this section was the handing of data through the Activities. In fact the **MainActivity**, the **GeneralInfoDisplayActivity** and the **GeneralInfoEditorActivity** continuously deals with Pet objects, and loosing the point during the development was easier than expected.



07.02.03 Notification system

All the notification system classes can be found in the package `com.oop1314.fido.notification`.

Since the first meeting of the team, the implementation of a local notification system was scheduled.
I started with the development of the first rudimental **ReminderService** class, as the Reminder section was part of my tasks. This class aim was to build a notification and to notify the *NotificationManager* using the extra of the *Intent* passed and the *pendingIntent* (necessary to open the **RemindersListActivity**). Once developed, I started working on the deletion, writing the code for deleting a notification in the **ReminderEditorActivity** (after, this piece of code will become the method *notifyEvent* in **AbstractEditorActivity**).
After the correct code was made for Reminder, I created the **DietService** and, afterwards, factored them in the **Template Method** for Notifications, with **AbstractService** class.

Unfortunately, when the phone used to reboot all the notifications were lost. In order to supply to this, the classes **MyBroadcastReceiver** and **MyBroadcastAlarmService** were created.
The first class extends *BroadcastReceiver* and has permissions to set notifications up after the re-boot of the phone. The second is the Service that **MyBroadcastReceiver** starts and wakes the *AlarmManager* up, passing it the notificableList of Event and Meal objects that the **DataHelper** computes.

The **INotification** interface is implemented both by Event and Meal model classes and helps the **MyBroadcastAlarmService** class to set notifications up.

07.01.04 Patterns

Understanding the Notification system template method

When I realised that I needed two different Services for Meals and Events, but with some common code, I chose to implement the Notification system through the usage of the **Template Method**.

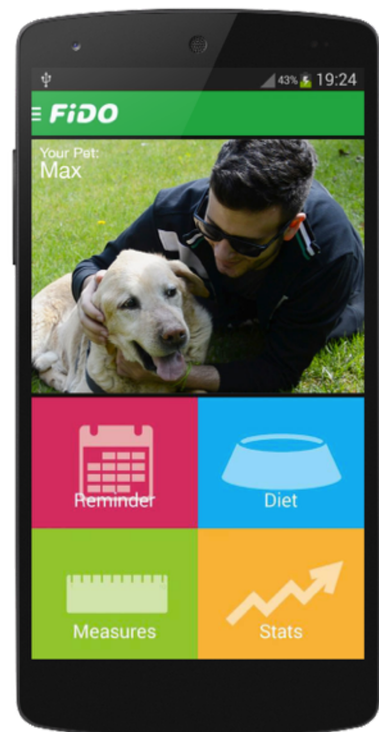
In the **AbstractEditorActivity** class, the method *notifyEvent* starts the Service: **ReminderService** or **DietService**, depending on the intent extra. In the main method of **AbstractService** class, the onStartCommand method, the local notification is built. When it is necessary to get specific info for the intent extra passed, the work is passed to the abstract methods, which are differently implemented in the two Service classes.

07.03 The common part

07.03.01 Main activity

The **Main Activity** is the meeting point for all app features. The main tasks that are entrusted to its activity are:

- loading the main Owner from the file, or showing the Welcome Activity if it doesn't exists;
- loading the Drawer data and updating it when necessary;
- associating the buttons with the relative Intents
- managing the adding, editing and deleting operations (with the help of MainController class).



07.03.02 Stats & Costs

Stats & Costs part is a totally readonly part that shows some statistics and the summary of the costs for the selected pet.

For this section we have used **AndroidPlot 0.6.0**, that is one of the possible choices for showing plots and graphs in Android.

In the plot section, we have two choices of visualizations: **Measures**, so you can see the evolutions of length, the height or the weight for the desired pet, or **Costs**, which shows a pie plot of the overall costs, divided into the Costs categories.

Next in the view there are some **statistics**: the pet's life and period since takeover and the list of total costs also here divided into categories.

The Activity manages the plots visualization; the dirty work of calculating all data is delegated to the Stats Controller.

DEPLOYMENT

08 Testing

The software has been tested throughout the development on many devices and emulators with Android 4.3 Jelly Bean MR2 OS (API level 18) and Android 4.1 Jelly Bean OS (API level 16):

- Android Virtual Devices (under Mac OS X 10.9 Mavericks 64-bit):
    - **ARM64 Emulator** (armeabi-v7a, 480x800px XHDPI, Cortex-A8 CPU - like Google Nexus S)
    - **Intel Atom Emulator** (x86, 480x800px XHDPI - like Google Nexus S)
- Real devices:
    - **Samsung Galaxy S3** (GT-i9300, 720x1280px XHDPI)
    - **Samsung Galaxy S3 Mini** (GT-i8190n, 480x800px HDPI)

The minimum SDK API level for the app is 16 (Jelly Bean) and the target API level is 19 (KitKat). These data are visible into the Android Manifest file (`AndroidManifest.xml`).

Two builds of the app have been distributed to a small circle of friends for a more effective testing and feedback.

09 Final considerations

The FiDO Team is **highly satisfied** of this project for some reasons.

The first is that the choice to develop this project on the **Android platform** is a challenge we have set ourselves to improve the way we develop in object oriented environments and also an opportunity to learn in detail how to develop in Android, a very important piece of the pie of the mobile operating systems market since it's installed on 78% of smartphones worldwide, according to IBTimes.com[1].

[1] *"Worldwide Smartphone Growth Estimates: Better Days Ahead For Windows Phone While Android And iOS Are Expected To Lose Market Share"* — International Business Times, Feb 28th, 2014 (bit.ly/1mNxrDl). Please note that, even if Android market share worldwide is 78%, only the 65% of the whole Android devices can run FiDO, since it's compatible from SDK 4.1 Jelly Bean. Source: *Android Dashboards*, Google (bit.ly/1fDoS6p)

09.01 Workflow

The workflow was fairly linear and approaches the proposed workflow published on February 11th, 2014 on Unibo EASI moodle platform.
The only changes made to the proposed workflow are the follows:

- Since we have decided to develop an app on an almost unknown platform, the first week of work has been focused for Giancecchi — who already has experience in mobile development for iOS — to understand the Android SDK, the structure and the features of its main components. During this week, Sapignoli worked on the review of the model and its writing as Java classes. Once this part has finished, the team merged the knowledges and started with the development of designated parts.

- The design analysis has remained almost unchanged, except for some choices regarding the data structures used (e.g. seeing what's better than Lists or Maps in certain situations) and for keeping or removing model fields and functionalities that are unnecessary or non-priority for the project.

- The implementation of the patterns was performed later — approximately at 40% of the work — since we aimed primarily to creating a basically working prototype of the app. Once we understood the various parts of MVC in Android and found the common parts of code, we proceeded refactoring classes and cleaning code using CheckStyle, FindBugs and PMD plugins.

09.02 Interactions & criticalities

There have been continuous interactions between the two components of the group aimed to discuss and implement the best design choices to be deployed.

The team worked separately on the code, using Mercurial as software versioning service and BitBucket as hosting, as recommended by professors.
Twice/three times a week the team met to work together on the common parts, to monitor the work's progress and to set some deadlines as goals to be achieved.

The critical points of the project that took more time were:

- for Nicola Giancecchi: the management of the I/O using DataHelper took a little bit more time for the construction and especially the testing, which involved the creating and reading the file and all the operations of addition, modification and deletion of individual objects.
  The use of AndroidPlot has requested a general learning of the platform and later other hours for its implementation.

- for Michele Sapignoli: the challenge to face a new OS, without having written a code line for mobile devices before, was exciting and hard at the same time. It took a while to get into the Android logic and to understand how the data flow and the entire system were managed. The hardest part was probably the development of the Notification system. First, I had to understand the possible implementation for

Android devices and then set task apart, in order to give a working solution for both Reminder and Diet objects. It was not so immediate, but I am particularly glad I have succedeed in developing it, as I am proud of the Reminder section.

10 Next steps

Although the software presented is stable and works well, there are some enhancements that are needed for presenting a more efficient and consistent management of data.

The next main step is implementing a physical **database** instead of the current .dat file. The project will be presented to professors Dario Maio and Annalisa Franco for the course of "*Basi di Dati*" (Databases). SQLite (`http://sqlite.org`) will be used as main DBMS.

Thanks to the use of **MVC** pattern, in principle will be necessary modifications only for the part of Controller — in addition to some minor graphical and model adjustments. The presentation will take place in the second half of June 2014.

Finally, we are planning to release the full app with database on **Google Play Store** during this summer and to develop an **iOS edition** of the app within this year.

11 Bibliography & credits

- Plotting library: AndroidPlot 0.6.0 - `http://androidplot.com`
- Date utilities library: Joda-Time 2.3, released under Apache License - `http://joda.org/joda-time`
- Cover and document photos: Michael Gil (`msvg`), Matther Rogers (`rogersmj`), Tom Gill (`lapstrake`), Dale Hichens (`gibzilla`) @ flickr, released under Creative Commons License.
- Special thanks to Matteo Pazzaglia, Daniele "Cuore di Panna" Fabbri, Thomas Righi, Guido Muscioni and Lorenzo Moraccini for testing this software… and for supporting us :)
- Thanks to Azienda Agricola Rodi di Roberta Sapignoli for the horses recordings.

12 Group contact informations

Nicola Giancecchi
UNIBO ID: 0000653628
Academic mail: nicola.giancecchi@studio.unibo.it

Michele Sapignoli
UNIBO ID: 0000653917
Academic mail: michele.sapignoli3@studio.unibo.it

App website

http://www.fidoapp.net/
nicola@fidoapp.net
michele@fidoapp.net

Android™, Google, Google Play Store and the Google logo are registered trademark of Google Inc.
IOS is a trademark of Cisco in the U.S. and other countries and is used under license by Apple Inc..
"Facebook" is a registered trademark of Facebook, Inc.. All rights reserved.