SDD

System design document for Learn Java

Version: 2

Date: 27/5-17

Author: Tobias Lindgren, Madeleine Lexén, Hanna Carlsson och Sara Kitzing

This version overrides all previous versions.

1 Introduction

This is a document which describes how the application Learn Javas system is designed. The specifications can be found in the requirements and analysis document.

1.1 Design goals

The application is designed for Android, and must therefore be able to run on Android. It should be able to run on Lollipop or higher. For Learn Java to be able to withstand changes, the code should be easily extendible. This means that the classes need to encapsulate behavior and be loosely coupled. These changes could be to the project scope and platform requirements. The application should also be able to compile code, therefore it requires a server.

1.2 Definitions, acronyms and abbreviation

Key: The correct answer
Level: A subcategory within a main category. Contains information about the specific topic and a question on the topic.
Map: The start page, a map over the different main categories
Read more: A page which shows information about basic programming in text form that the user will learn throughout the different worlds. It can be accessed from the menu in all views.
Boss: The last level in a main category which summarizes what the previous levels has taught the user.
Fill in the blanks: A question where the user will get a number of sentences with blanks that they will have to fill in themselves.
Write code: A question where the user will get a task where they need to write code
Toast: A rectangle with text that appears at the bottom of the page

2 System architecture

2.1 Overview

There are two machines involved in this system; the server, here represented by the computer running the program, and the phone the app is running on, either a virtual machine or a physical phone.

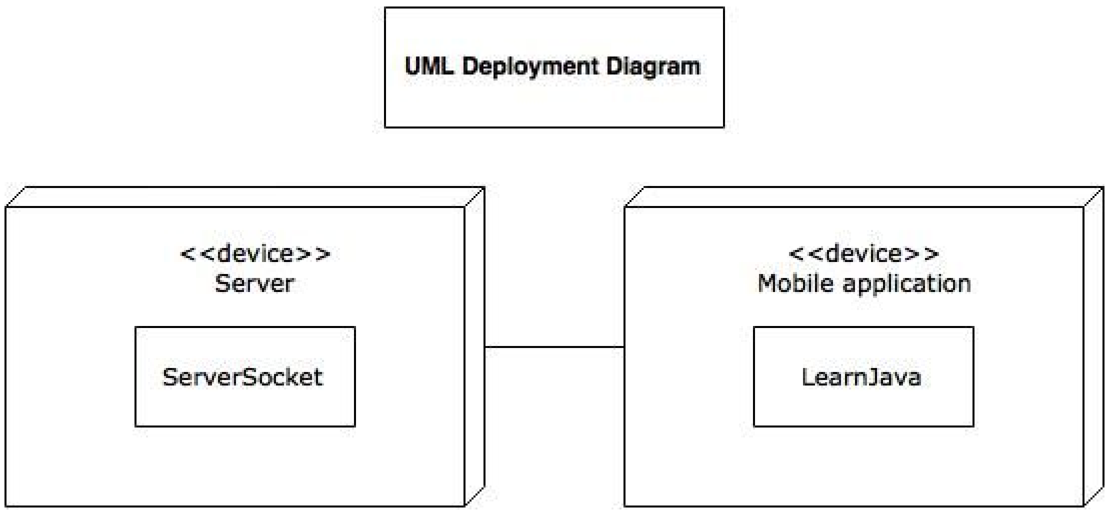

Figure 1: The applications UML deployment diagram

2.2 General observations

Server

The program needs to be able to execute code written as user input in the Write code-queries. Since the program is an Android app this is not possible to do on the actual device, this is why the server is implemented. The server is based on a ServerSocket, which is a class handling requests. The class waits for a requests, performs an operation and returns the result depending of what is needed [1]. A string with the user input is sent from the app to the server, which executes the user code and sends back the result as a string. The server also handles necessary operations such as setting up and closing the streams. The server in this project is not on an actual server since we did not have access to one, therefore you need to run the app on the same computer as you are running the server. The server communicates with the app through the *WriteCode*-query. This class creates the server-object and handles sending and receiving data from the server.

Executing code with BeanShell

BeanShell is an embeddable library, through this you can dynamically execute Java code at runtime [2]. In this application it is used together with the server when compiling and executing the code from the user input. The only class used in BeanShell is the Interpreter. The method eval is used to execute a String that comes from the servers inputstream. The interpreter-output is then sent to the file compiledCode. The executed code gets shortened before sent back, so the server only gives the latest result.

User and statistic

The app has a login function and therefore a user. Each user is an object of the class User, and is saved as a bin file each time the app is closed. When a user logs in the users object is loaded from the bin file through the AccountManager singleton class, which handles the login and add new user methods. Every user object holds a reference to a statistics object where the statistics is saved. The statistics model works as a database but instead of saving the data to a real database, the data is stored in lists.

Query

The app has a few different variation of questions, and these have a common superclass Query. This is so polymorphism can be utilized when delegating the checking of the answer to the right type of question. It also diminishes code duplication.

Category and levels

The game is divided into four categories, “Primitive types”, “boolean-logic and if-statements”, “for- and while-loops” and “arrays”. Each category contains five levels of three different types, “Multi choice”, “Fill in the blanks” and “Write code”. When accessing the game for the first time, only the first category and first level is unlocked. To unlock the next level you have to complete the previous level first. When all levels in a category have been completed, the next category is unlocked.

Each level is a Query-object, which is an abstract class. It has three subclasses, MultiChoice, ModelFilllBlanks and ModelWriteCode. The Query-object takes a List of Strings as it’s parameter. From this list it will set the question, answer, hint, info and heading of the level.

Depending on which type of question it is, it may set one more attribute to the object. If it is a Multi choice-object it will also set the alternatives. If it is a Write code-object it will set an alternative answer. Since we give the possibility to show the answer while trying to solve the question, we have to show them an alternative answer for Write code. This is because the check answer method only compares the output of the compiler and not the entire code the user should answer with.

2.3 MVP

Model-View-Presenter is implemented by having three main packages. A layout, a controller and a model package. MVP is a part of the Model-View-Controller, but a more common pattern to use when dealing with web and mobile applications [3].

The layout package is the view part of MVP, and it holds all the Android .xml files. In the controller package are all the activities and their help classes. The activities acts as the presenter in this case and handles all the communication between the view and model. The model package handles the logic and storing of data. It has no knowledge of any other package, and only communicates within the model package. There are no communication from the model to the presenters.

2.4 Domain model

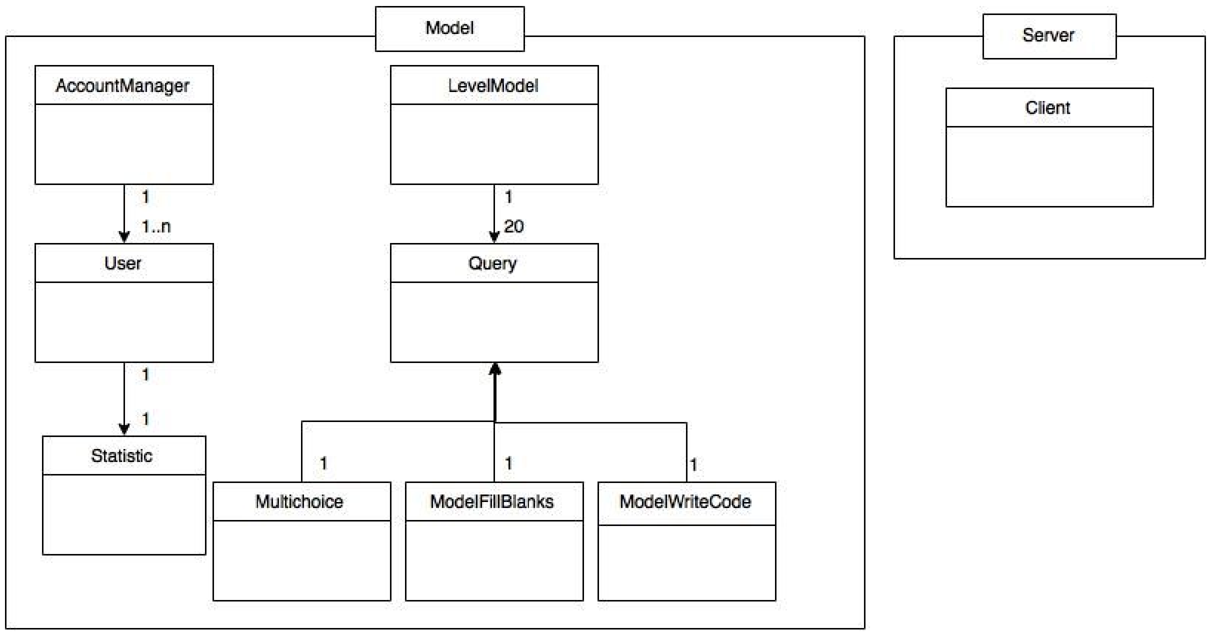

Figure 2: The application's domain model

LevelModel: This class is responsible for maintaining and storing the hashmap with the level objects.
Query: This class is an abstract superclass to the question classes, and gives them a common interface. It is also responsible for keeping track of the questions, and maintaining them.
Multichoice: This class handles the logic for the multichoice question, and corrects the users answer.
ModelFillBlanks: Handles the query where the user gets to fill in the words that are missing in a text.
ModelWriteCode: Handles the query where the user gets to write its own code.
AccountManager: Handles the login and add user functions along with loading and storing the user objects.
User: Handles the updating of the user and holds a statistics object which calls the save statistics methods.
Statistics: Stores the time it takes to finish the question, how many hints needed and if you looked at the key.
Client: Handles receiving the data from the server, compiling the code and sending it back.

3. Subsystem decomposition

3.1 Learn Java

Learn Java is decomposed into four packages: model, controller, layout and service. Model, controller and layout are implemented using the MVP-pattern, service handles reading and loading files and connection to the server. All the classes in the services package communicates with the model through the controller-classes.

3.1.1 Model

The model package handles the logic behind the application. It does not have any connections of its own, the controller communicates with it.

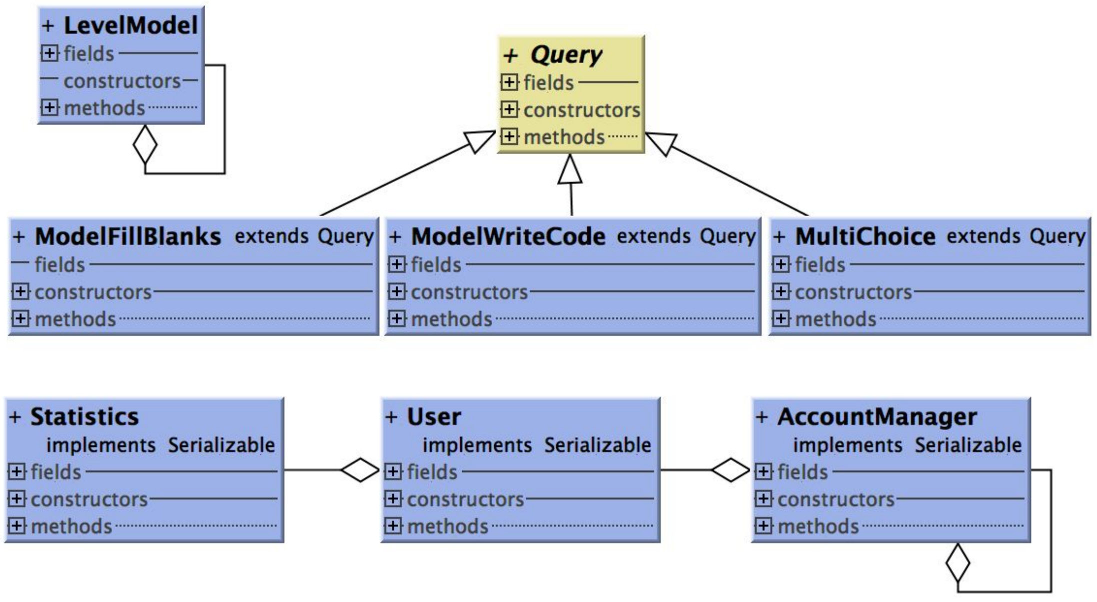

Figure 3: UML-diagram of Model package

3.1.2 Controller

The controller package handles all the communication in the program. It has listeners on buttons in the layout and sends the information to model-objects. It also handles the communication between the service package and the model.

3.1.3 Layout

The layout package contains all the .xml files. Each component in the layout has an id which is coupled in the controller package.

3.1.4 Service

The service package holds all the classes that are not connected to MVC. It handles tasks such as reading a file, getting images and setting up and connecting to the server.

Server

The server takes care of compiling and executing code. See *2.2 General observations* for a more detailed description.

FileReader

The FileReader reads the information for all levels (info, question, answer, hint, header and alternatives/alternative answer). The information is stored in a .txt-file and the method for reading reads the file by line, where the different texts are stored on separate lines (question on row 1 and answer on row 2 of the .txt-file).

ImageHandler

The ImageHandler loads and stores images in the internal storage. The images are connected to the user by having the same name as the username of the user.

UserFileReader

The UserFileReader saves and loads the AccountManager singleton. It saves the object in a directory, and uses an ObjectOutputStream to save the object to the directory. To load the object it uses ObjectInputStream.

3.1.5 Diagram

Dependency analysis

The applications dependencies are as shown below. There are no references to the controller, or between two packages where none of them is the controller. Therefore the controller is encapsulated and the application is following MVC. There are no circular dependencies between the packages, however circular dependencies do appear in the controller package (see appendix). This is not optimal but there was not time enough to solve the issue.

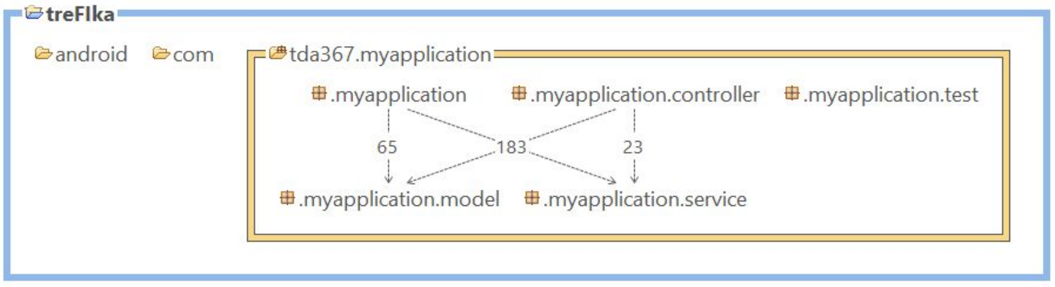

Figure 4: The stan dependencies between the packages

Flow diagram

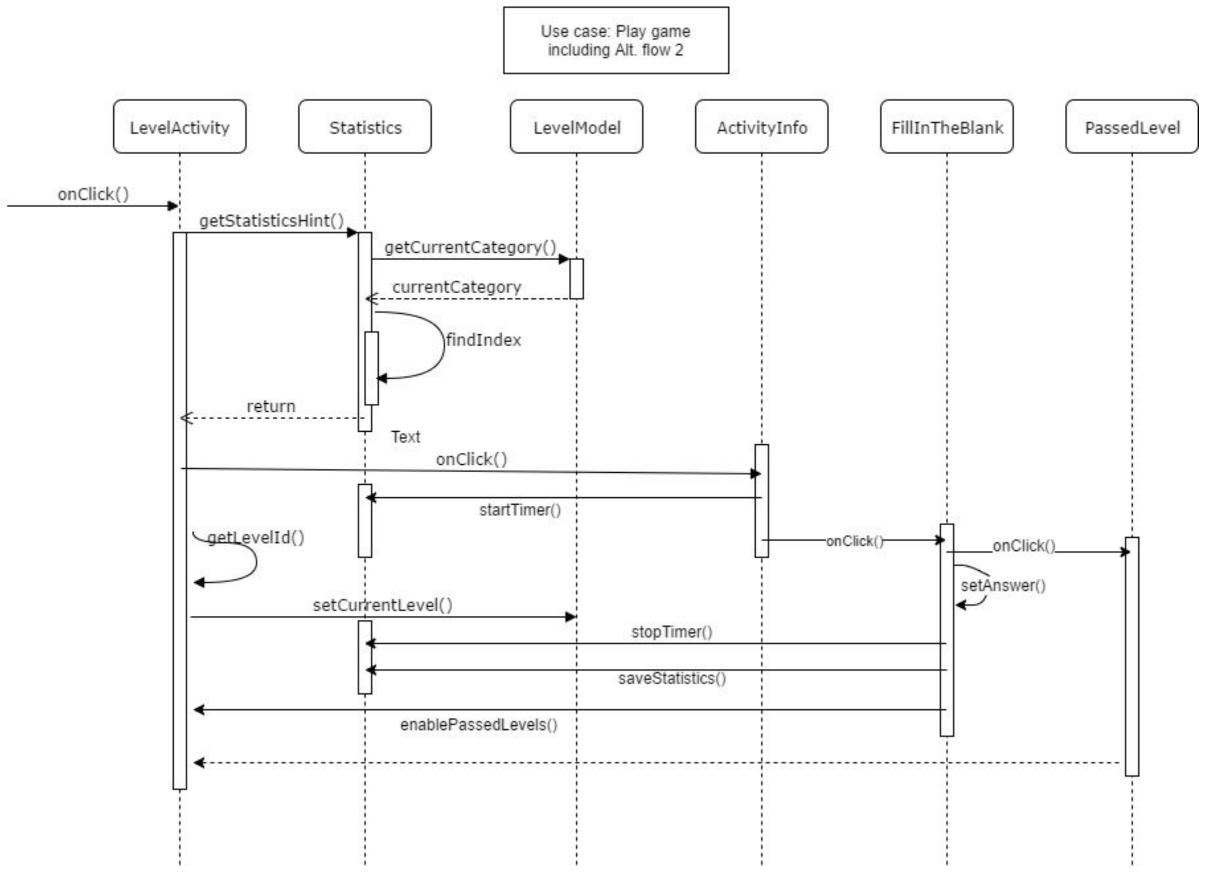

Figure 5: Use case Play game with alternate flow 2.

3.1.6 Quality

All of the relevant methods in the model-classes is tested, as well as some in the service-package.

List of tests

- AccountManagerTest
- ExampleUnitTest
- FillBlanksTest

- LevelModelTest
- MultiChoiceTest
- ModelWriteCodeTest
- ServerTest
- StatisticsTest
- UserTest

Quality tool report

Some “System.out.print” can be found in the catch-blocks of the server-class which makes the code less useable. The reason for this is because there is no need for fixing this problem even though it limits the applications complexity.

The code also contains some “NullPointerException”, this could hide normal errors and instead causing errors that might give worse consequences [4]. This is affecting the reliability of the code. However the reason for the application throwing these errors is unknown, but the application still works as expected, therefore the exceptions remains.

3.1.7 Concurrency issues

Two threads are necessary in Learn Java, one for the main application and one for running the server. However when the “server thread” runs, the main thread waits for it to be done. Therefore the concurrency issues will not be a problem in this application.

3.2 Client

The server-client is a “one-class-machine”, it contains a few methods for setting up, connecting and closing the server as well as methods for the actual assignment; compiling the code. It also has a class for creating and running the client. The server does not need to save any data from time to time, and does not contain any packages and therefore no dependencies to analyse. The test of the server goes through the main application.

However there are some “System.out.print” in the code which might not be optimal for the applications usability. However, the reason is because there is no “real server”, therefore there needs to be a way to see if the server-program actually is running and doing what it should. Since the program is just a class and does not have any real view to display things, the easiest way to accomplish getting status updates from the server is through the monitor of the program.

4. Persistent data management

The application data is mainly stored as a bin file. This file contains an AccountManager with a hashmap that holds each of the applications usernames as the key and the corresponding User-objects as the value. Each User-object holds the information associated to the specific user as well as a Statistic-object containing an arraylist with statistics for each level. However, the pictures are saved in a directory where each picture is related to its User through the user’s name.

The text for all queries is saved in text files, which are read when creating all the level objects when the program is started.

5. Access control and security

NA

References

[1] “Class ServerSocket”, *Java Platform SE 7* [Online] Available: http://docs.oracle.com/javase/7/docs/api/overview-summary.html [Retrived: 26 May, 2017]

[2] “What is BeanShell?”, *BeanShell* [Online] Available: http://www.beanshell.org/intro.html [Retrieved: 21 May, 2017]

[3] F. Cervone, “Model-View-Presenter: Android guidelines”, *A Medium Corporation* [Online] Available: https://medium.com/@cervonefrancesco/model-view-presenter-android-guidelines-94970b430ddf [Retrived: 25 May, 2017]

[4] QAPlug PMD plugin in Android Studio and IntelliJ

Appendix

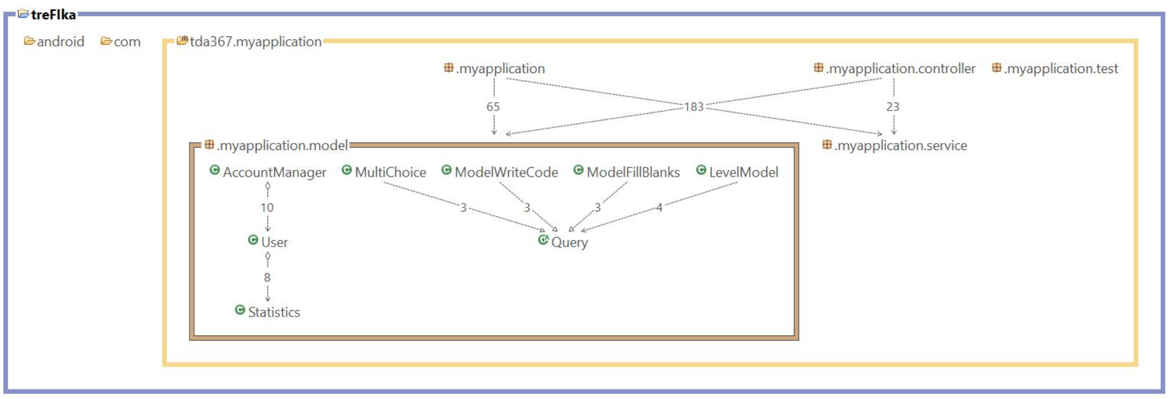

Figure 6: Dependencies between the classes in the model

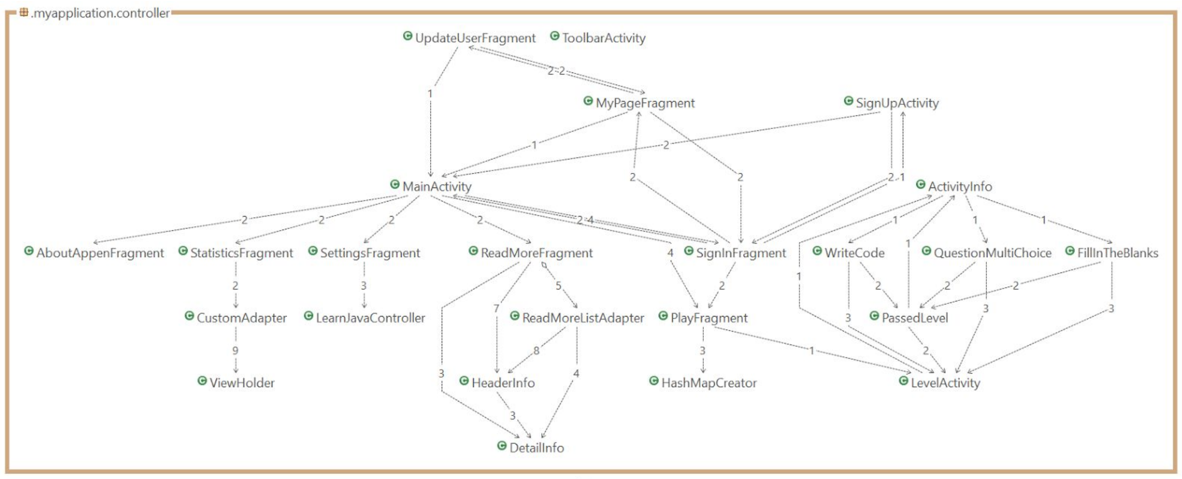

Figure 7: Dependencies between the controller classes

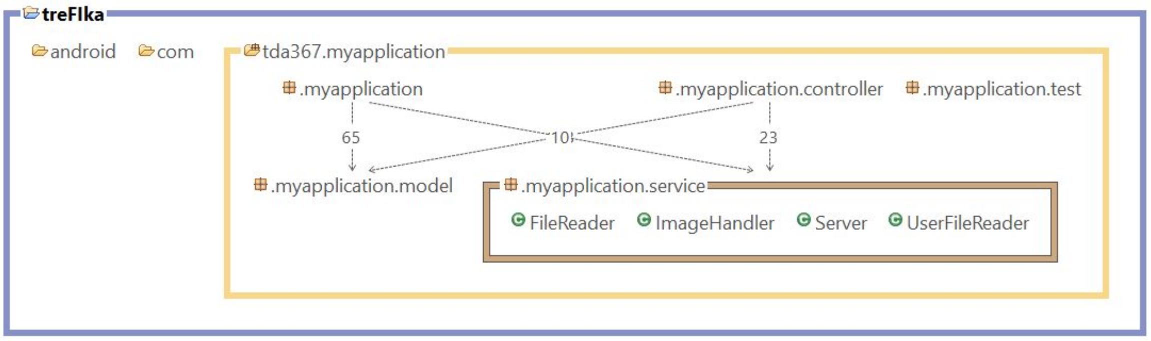

Figure 8: Dependencies between the services classes