Mathinator
Software Architecture Document

Version <1.1>

Revision History

Date	Version	Description	Author
<28/11/2016>	<1.0>	<Creation of the SAD>	<Lamm, Hug, Saupp>
<27/12/2016>	<1.1>	<Adapt the Technology + Database Diagram>	<Lamm, Hug, Saupp>
<26/5/2017>	<1.2>	<Add Use Case>	<Lamm, Hug, Saupp>

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

This document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made on the system.

1.2 Scope

The scope of this SAD is to provide the architecture of the Mathinator project. Affected are the class structure, our use-cases and the

1.3 Definitions, Acronyms, and Abbreviations

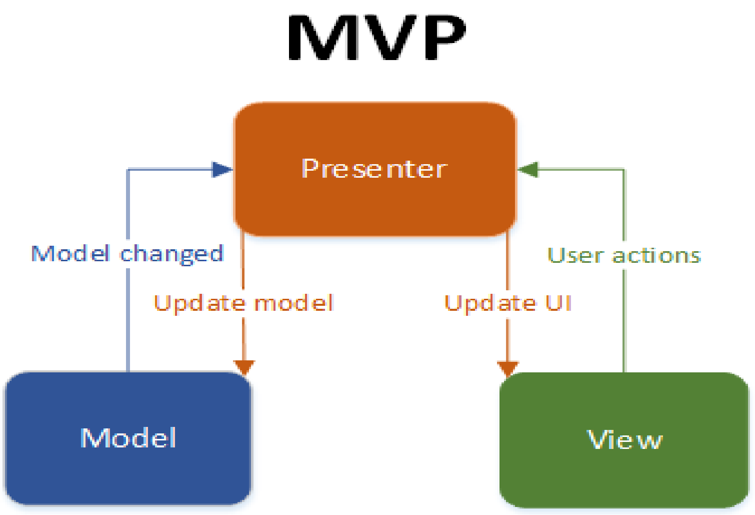
1.4 References

Document	Where to find?
Blog	https://mathinator.tobiaslamm.de
Github	https://github.com/SaschaHug/Mathinator
Use Case 1 “take a picture”	https://github.com/SaschaHug/Mathinator/blob/master/Use%20Cases/1_UC_Mathinator_Take_A_Picture.pdf
Use Case 2 “view history”	https://github.com/SaschaHug/Mathinator/blob/master/Use%20Cases/2_UC_Mathinator_View_History.pdf
Use Case 3 “show tour on first start”	https://github.com/SaschaHug/Mathinator/blob/master/Use%20Cases/3_UC_Mathinator_Delete_Entry.pdf
Use Case 4 “enable user to delete entries”	https://github.com/SaschaHug/Mathinator/blob/master/Use%20Cases/4_UC_Mathinator_Use_Manual_Calculator.pdf
Use Case 5 “do manual calculations”	https://github.com/SaschaHug/Mathinator/blob/master/Use%20Cases/5_UC_Mathinator_Show_tour.pdf
Use Case 6 “accept picture”	https://github.com/SaschaHug/Mathinator/blob/master/Use%20Cases/6_UC_Mathinator_Accept_Picure.pdf

1.5 Overview

The rest of the document is separated into ten different chapters.
Chapter 2 will describe what software architecture is for the current system, and how is represented. Chapter 3 will cover the software requirements and objectives that have some significant impact on the architecture. Chapter 4 lists use cases/scenarios from the use-case model which have significant impact on the architecture itself. Chapter 5 describes the architecturally significant parts of the design model, such as its parts Controller and Model. Chapter 9 is a description of the persistent data storage perspective of the system.

2. Architectural Representation



3. Architectural Goals and Constraints

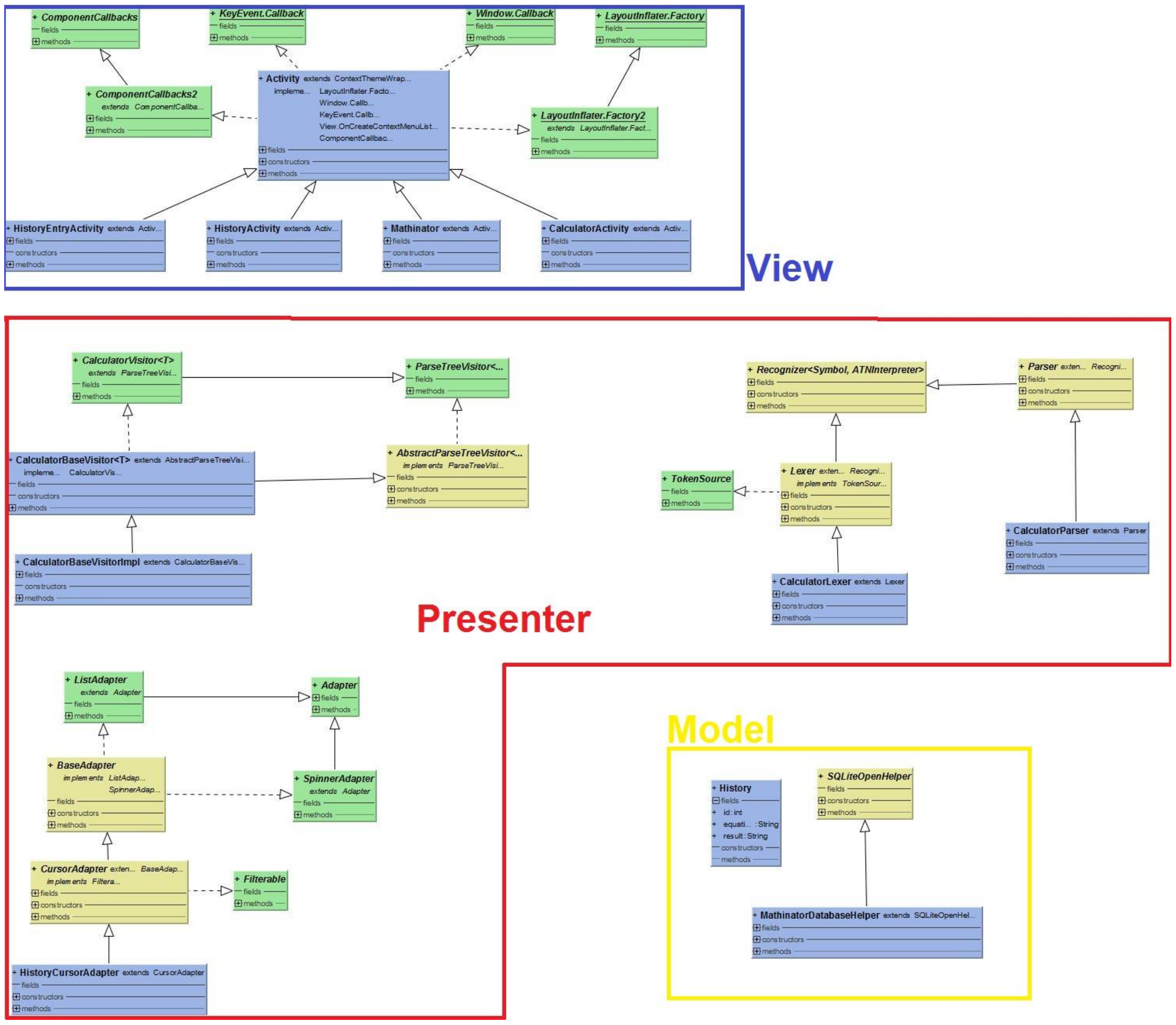
In Android you don't have MVC, but you have the following:

- You define your user interface in various XML files by resolution, hardware, etc.
- You define your resources in various XML files by locale, etc.
- You extend clases like ListActivity, TabActivity and make use of the XML file by inflaters.
- You can create as many classes as you wish for your business logic.
- A lot of Utils have been already written for you - DatabaseUtils, Html.

4. Use-Case View

n/a

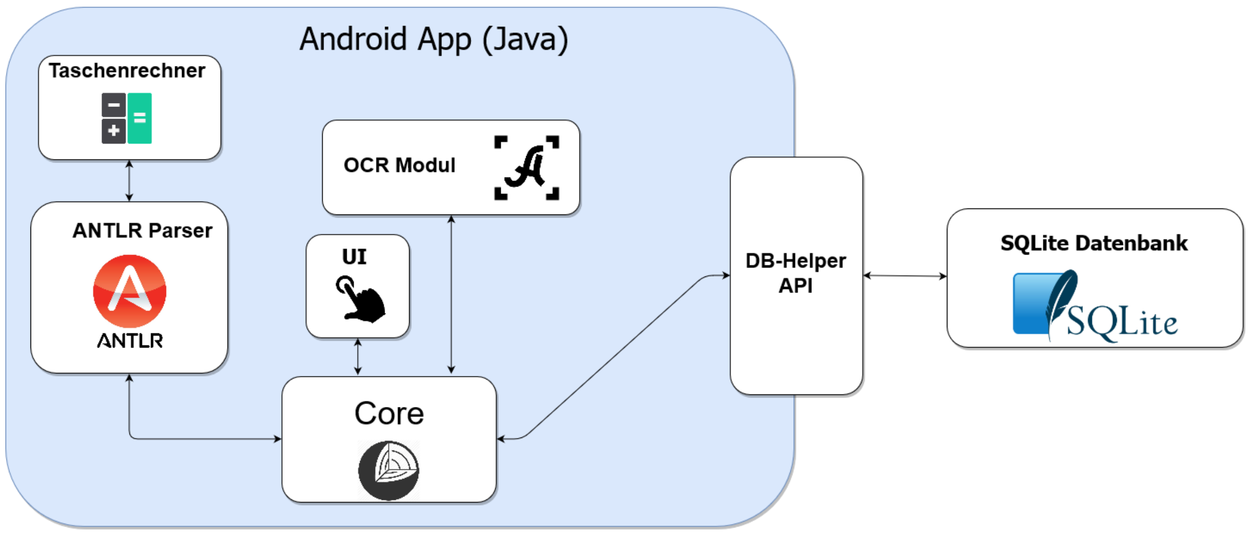
5. Logical View


View
ComponentCallbacks
KeyEvent.Callback
Window.Callback
LayoutInflater.Factory
ComponentCallbacks2 extends ComponentCallba...
Activity extends ContextThemeWrap... impleme... LayoutInflater.Facto... Window.Callb... KeyEvent.Callb... View.OnCreateContextMenuList... ComponentCallbac...
LayoutInflater.Factory2 extends LayoutInflater.Fact...
HistoryEntryActivity extends Activ...
HistoryActivity extends Activ...
Mathinator extends Activ...
CalculatorActivity extends Activ...
Presenter
CalculatorVisitor<T> extends ParseTreeVisi...
ParseTreeVisitor<...
CalculatorBaseVisitor<T> extends AbstractParseTreeVisi... impleme... CalculatorVis...
AbstractParseTreeVisitor<... implements ParseTreeVisi...
CalculatorBaseVisitorImpl extends CalculatorBaseVis...
Recognizer<Symbol, ATNInterpreter>
Parser exten... Recogni...
TokenSource
Lexer exten... Recogni... implements TokenSour...
CalculatorLexer extends Lexer
CalculatorParser extends Parser
ListAdapter extends Adapter
Adapter
BaseAdapter implements ListAdap... SpinnerAdap...
SpinnerAdapter extends Adapter
CursorAdapter exten... BaseAdap... implements Filtera...
Filterable
HistoryCursorAdapter extends CursorAdapter
Model
History
id: int
equati... : String
result : String
SQLiteOpenHelper
MathinatorDatabaseHelper extends SQLiteOpenHel...
fields
constructors
methods

6. Process View

n/a

7. Deployment View



7.1 Description

The Application is structured into different pieces, each of which are designated to implement a specific functionality.
The ANTLR Runtime is generated by a Parser Generator which in turn relies on a Context-Free Grammar File that describes how valid Equations are be structured. The Parser Routines are auto-generated and thus, changing the grammar / adding new features can be done easily since the code only needs to be modified in one place.
The OCR Module is in Control of reading and processing the users handwriting, which is matched against an example-set. A neural Network is used to compare the two and find the closest match.
The UI lets the User control / start these Actions.
The DB-Helper API is to ensure that all CRUD Operations are bound to a Singleton Instance of the SQLite Database, so that at any point, there may only be one instance of the Database across the Application.

8. Implementation View

n/a

9. Data View (optional)

We use a SQLite Database to persist Data to the Android Device. Since there is no need for relations and foreign Keys, we only have one Entity, which can be found here.

10. Size and Performance

n/a

11. Quality

n/a

12. Patterns

In larger software projects, patterns will make your life easier (that is, if you use them properly). They will speed up the development process by providing tested and proven development methods.

In our project we used a so-called “Interpreter Pattern” which is supposed to implement a domain specific language (in our case a calculator interpreting arithmetic statements).

There are serveral steps involved in using and creating such a pattern:

- Design a grammar for said language
- Each production (e. g. PlusOperation -> Number ‘+’ Number) needs to be mapped to a function
- All of these functions are specified by a composite pattern (We used an Interface that defines the relevant methods)
- The context of each method must then be interpreted. This is done by a Visitor and a VisitorImplementation Class (which implements the methods given by the interface)
- The context is then interpreted until all nodes of the syntax-tree are resolved (and thus the result is calculated)

We used this pattern because is makes implementing a calculator a lot easier. When you have to solve a lot of similar problems, it makes sense to define a interpreting language for it (e. g. regular expression or arithmetic operations). Also, we don’t have to rely on complex algorithms that sort out operator precedence for us. We are also able to extend the grammar quite easily using this pattern.