Masaryk University
Faculty of Informatics

Cross-platform development of smartphone application with the Kivy framework

Master thesis

Bc. Ondřej Chrastina

Brno, Autumn 2015

Declaration

Advisor: Mgr. Juraj Michálek

Acknowledgement

I would like to thank my supervisor Mgr. Juraj Michálek for his valuable help, comments, suggestions and time he spent helping me with this work.

Abstract

The purpose of the thesis is to analyze Kivy Framework (www.kivy.org) and summarize its capabilities. The main goal is to deliver sufficient information about the framework to decide on the possibility of its use it in commercial development. The practical part is devoted to a prototype application development that is used as a proof of concept. The prototype helps to identify potential problems during the implementation and deployment.

The text is structured in three parts. The first part familiarizes the reader with the framework and dependent technologies. The second part is concerned with using the collected information and uses that as a base for developing the application. The last part validates the strengths and weaknesses of the framework during the application's development process and describes the problems found during the implementation.

The results of this research are used as a decision point on whether the Kivy Framework is suitable for commercial development in Y Soft Corporation, a.s. The benefit of the thesis is that the comprehensive description of the Kivy Framework can be beneficial not only to Y Soft Corporation but to any potential programmer who is concerned with this framework and its potential use.

Keywords

Kivy Framework, Python, KV language, Y Soft Corporation, a.s., YSoft Safeq, REST, multiplatform development, user interface, Raspberry Pi

Contents

List of Figures

Chapter 1

Introduction

Today, the market is filled with many platform and operating systems. When developing software, it is often necessary to implement applications for every single system and support them separately. However, a different approach exists. Implementing application once and finding out how to run the application on the majority of platforms on the market. One of the frameworks providing a support for this development approach is the Kivy Framework. It allows to implement an application once and then create a distributable package for the majority of operating system.

This thesis describes how to create an application using the Kivy Framework, how creating distributable packages for the supported platforms works and the inner workings of the framework itself.

1.1 Thesis goal

The goal of this thesis is to analyze the Kivy Framework. This analysis should comprise the complete set of information for potential developers. The following text includes frameworks architecture, capabilities, strengths and weaknesses. This thesis in not a transcription of the official Kivy documentation. It meant as a practical guide, describing best practices, guidelines and tips on how to use the framework and how to develop maintainable programs efficiently.

All the instructions are validated on an application that was developed in parallel as a proof of concept. It is a multi-platform client that uses a Terminal Server REST API developed by Y Soft and described in section 3.2. The application provides management of print jobs in the printer network. This interface was primarily purposed to serve print job data to applications in a computer network. It is currently used by a web application that allows it to manage print jobs. This thesis is a research in whether Kivy is sufficient to provide a user-comfortable application for print job management.

The combination of research and development cover this framework from a theoretical and practical perspective. It includes some significant topics that aim on the community, services and development process and the potential of this framework.

1.2 Kivy

Kivy is an open source community project for multi-platform development supporting most of the desktop and mobile platforms as Windows, Linux, OS X, iOS and Android. It also allows to run applications on Raspberry Pi[1]. The framework is written in Python and it can be developed on all three supported desktop operating systems, on Windows, Linux and OS X.

The first version of Kivy was released on February 1st, 2011. The current version is 1.9.0, which was released in April 3, 2015 2.1. The framework has a complete support for Python version 2.7.x. Development of a version with full support of Python 3.x, which is not fully backward compatible with the previous major version [20], is in progress. This is possible largely because of $5000 grant from PSF[2].

Kivy is published under the MIT license[3] (version prior 1.7.2 are under LGPL[4]), so it is business friendly and it is possible to use it in commercial sphere. The main authors are Thomas Hansen, Christopher Denter, Edwin Marshall, Jeff Pittman and Brian Knapp [8]. It is not only one a core library. It also cooperates with sister projects such as Python for Android, Kivy for iOS, Kivy Designer, Buildozer, and others [10, 8].

This was only a brief review of Kivy, mostly from an organizational point of view to get an idea before the actual research. Tech-

1. Credit-card sized computer that plugs into a monitor or TV.
2. Python Software Foundation - Nonprofit organization, devoted to the Python programming language.
3. The type of Open source license
https://opensource.org/licenses/MIT
4. GNU Lesser General Public License
http://goo.gl/AL2Psj

nical aspects and analysis are described in section Research 2.

1.3 Y Soft Corporation

Y Soft Corporation is the Czech company, founded in Brno in 2000. It focuses on printing services. After fifteen years of improvements, Y Soft employs nearly three hundred experts mostly situated in its Brno headquarters. Today, Y soft aims on two main sectors in printing. The first one is 3D[5] printing, which offers simple solutions for creating things using printers. The product is called bee3D. This sector is not an interest of the thesis.

This work is focused on the second branch of Y Soft interest—YSoft SafeQ—a system for managing and monitoring print, copy and scan jobs in computer networks. This program helps customers reduce the cost of print management, increases document security and offers reports about all printing operations. It comprises a complex set of features such as authentication, rule-based printing, mobile print, scan management, billing functionality, reporting, etc.

Thanks to its double-digit revenue growth, Y Soft needs to follow modern trends [1] to keep its prestige. This is also the main reason for this research, because Y Soft is contemplating creating a mobile client of its application. Currently, it is possible to send a document or image to a printer with mobile device via email, or a web page. However, the YSoft SafeQ software offers much more functionality than simply sending documents to printers. It would be difficult to provide all the other functionality by mobile devices without a native[6] or hybrid application[7] [21].

1.4 Thesis structure

Thesis is structured in three main sections. The first section is Research 2 and it is about the Kivy Framework and its capabilities and

5. Three-dimensional space.
6. Program that does not require any external support (i.e. emulation) for running.
7. Application that is written with web technologies, and provided as native application using an embedded web browser.

architecture in general. The second section is Implementation 3 and it covers a practical implementation of an application using the Kivy Framework. It contains functional requirements, guidelines and information on how to implement these. It also involves deployment on various platforms. The last section is dedicated to Validation 4. The results are then compared with expectations. At the end of the section, there is summary containing pros and cons of Kivy development for this particular type of application.

At the very end of the thesis, there is an overall evaluation of development using the Kivy Framework and the final application.

Chapter 2

Research

This chapter deals with Kivy in general. It contains a description of the framework, its architecture and purpose. The main goal of this part is to convey the main idea of the framework and its capabilities. It also contains information about the authors and the organization standing behind the framework. The framework is open source [11], so the community is a really important part of it. That is why there is a section about community, support and whole organization, about its works and description about communication channels in section 2.1.1.

It is good to know the general idea of the development process. When a programmer makes this initial step and wants to make the application in a maintainable way, it is necessary to follow certain rules to make the final application testable, extendable. The sections that follow in this chapter contain best practices and specific guidelines. Information that is not covered by the official documentation is written in the form of tips that are mainly based on practical experience with development of the sample application.

Before the text starts digging down to various technical themes, let's summarize all the possible sources that Kivy offers for learning. Since it is an open source framework which was launched only four years ago according to the changelog [1], there are not many sources to learn from. The main resource is the documentation [7] which includes an API reference. When a problem comes up during development, the best way to solve it to check the support google group [4]. The group is called "Kivy users support"[2]. This is usually the best way how to solve any issues that come up during the development

1. Kivy changelog: http://kivy.org/#changelog
2. Kivy support forum: https://goo.gl/yP5fOD

phase. There are nearly five thousand of threads and if not one covering the programmer's problem, it is possible to create a new one and ask for help. Be aware that official documentation is connected with the last developed version. The documentation for stable and released versions are versioned using the Git versioning system [3].

When a programmer starts writing the application, it is good to know the basic concepts of the framework. An effective way on how to get knowledge about the framework is to go through the tutorial, which is part of the documentation. This tutorial is very basic. A more detailed one is the Kivy Crash Course by Alexander Taylor[4]. This tutorial shows how to work with Kivy objects, describes its capabilities and explains particular behavior, e.g., setting the size of widgets.

2.1 Kivy Framework Development

First version of Kivy was released February 1, 2.1.2. Since that version, the application was multi platform. Base concept of framework does not changed until the current version 1.9.0. First ten minor versions released during the year 2011 was focused on architecture. Kivy did not offered many ready made widgets at that time, just a basic set. The main focus was on creating a robust and reliable core. After the version 1.1.0 Kivy started to release new widgets to satisfy needs that require modern UI building. In version 1.8.0 and 1.9.0 was released many small improvements for widgets inspired modern trends as mobile development. The highlights of all versions are summarized in following time-line 2.1. Kivy Framework currently supports Python version higher than 2.7 and Python version higher than 3.3 for development. Full functionality for creating a distributable package is released for Python version higher than 2.7. The current official recommendation is to develop in Python 2.7+ or use a conversion between two version branches to achieve ability to package application written in Python 3 [7, 20].

3. Sources for the last released version of Kivy Framework: `https://goo.gl/sxHKUG`
4. Youtube channel containing Kivy Crash Course: `https://goo.gl/uXgHRa`.

February 1, 2011	**1.0.0** Running application for Windows, OS X, Linux and Android. Multitouch support using platform specific API. Graphics API using OpenGL ES 2.0. Released with complete documentation and API examples.
February 10, 2011	**1.0.2** Performance improvements, reduced memory requirements and simplified callback logic.
February 22, 2011	**1.0.3** Fixed bugs in packaging process. Added new widgets. Shader functionality extension.
March 20, 2011	**1.0.4** Widget set extension. Generated documentation with Programing guide.
April 14, 2011	**1.0.5** OS X packaging improvements. New template support for KV language with recursion detection.
June 3, 2011	**1.0.6** Android deployment improvements. Graphic rendering boost.
July 16, 2011	**1.0.7** Enhanced graphics and core functionality.
October 24, 2011	**1.0.8** Introduced virtual keyboard widget. Ability to detect platform.
November 14, 2011	**1.0.9** Video enhancements. Prevention of memory leaks.
February 13, 2012	**1.1.0** Markup rendering for all text widgets. New KV language parser.

February 15, 2012	**1.1.1** Bug fix for audio loader.
April 2, 2012	**1.2.0** Application monitoring module. Bug fixing for packaging.
June 19, 2012	**1.3.0** New UI theme. Optimized handling for platform specific button actions.
February 9 2012	**1.4.0** New Programing guide. New form widgets.
September 30, 2012	Properties extension for essential geometric shapes.
December 9, 2012	**1.5.0** Scale independent and density independent units.
December 13, 2012	**1.5.1** Bug fixing release. Line instruction fix.
March 10, 2013	**1.6.0** Scheduling methods calls. Widgets improvements for sophisticated content.
May 13, 2013	**1.7.0** Padding and spacing improvements used cascade style concept of value list.
May 28, 2013	**1.7.1** Fixing development blockers.
August 4, 2013	**1.7.2** Memory leak preventions.
January 30, 2014	**1.8.0** Base refactoring. Focused on widget properties improvements. New mobile centered Programming guide.
April 3, 2015	**1.9.0** Minor widget properties. Documentation improvements.

Table 2.1: Timeline

2.1.1 Kivy Organization and Sister Projects

Authors of the Kivy Framework call themselves the “Kivy Organization”. They are a non-profit membership organization developing this multi-platform product. All contributed code is provided as Open Source. The Kivy itself is under MIT license (starting 1.7.2) 2.1 and LGPL 3 license for the previous versions. The framework can be used in commercial products. Kivy is a community project, led by professional developers as is written on the Organization section of the official site [8]. It is possible to contact them and ask for help or share new ideas. The developers also participate in conferences and present their project in “Kivy talks”. As was mentioned, this project is open source and it is possible to contribute code right into development after the acceptance the contribution request.

This group, besides the main project, develops sister projects that help cover all the announced functionality. These projects are mainly used to provide functionality for all the supported platforms. The main task is to use system-dependent API to process requirements from Kivy. The main projects are Python-for-android, Kivy-ios, PyJNIus, PyOBJus, Plyer, Buildozer and Kivy Designer. All of the mentioned projects are released under the MIT license. These projects cooperate to create a distributable package. The detailed description of the particular projects is in following paragraphs. Cooperation of these project is summarized in figure 2.1

Python-for-android Python for Android is a project used for Android platform packaging. The main goal of this set of scripts is to create a standalone Android application package (APK)[5]. It is possible to place the package on an official repository provided by Google, Inc. It is also possible to place it on any other compatible repository. The result of the packaging is the same as is produced by native application development. It can be also used for commercial purposes.

Kivy-ios A compiler that manages the creation of the Xcode project [23]. This is the only way to create applications. It is also the only op-

5. Package format used for Android application distribution. It is based on the JAR format.

tion how to run the code on an emulator during the development and how to export the application into a distributable package. To be able to create it, the developer has to have a device running the OS X operating system [19] and be registered like a iOS developer.

PyJNIus A Python module that is used to access Java classes as Python classes using Java Native Interface (JNA)[6] The provided connection is used to join Python code and platform specific functionality i.e. mobile vibrations, because the interface for this parts of Android is written in Java. In fact, this module uses Cython to call C or C++ code from Python and then JNA which allows for running C or C++ code in Java [16].

PyOBJus Another Python module to make a link between platform specific interface and Python code. It is an alternative to the PyJNius module for OS X and iOS operating system. These systems provide access for functionality written in Objective-C. PyOBJus uses reflection of the Objective-C runtime to access Objective-C classes from Python [17].

Plyer Player is a platform-independent API that allows to access platform-specific functionality summarized in table B.1. It is a proxy, or a distributor of method calls between Python and specific platform API. It is an interlayer between Python for android, or Kivy-ios and specific programs that provide that functionality as PyJNIus for Android, PyOBJus for iOS and common libraries for Windows, OS X and Linux [13].

Buildozer A tool that works as a packaging provider. It is basically a set of scripts that allows packing applications on all platforms using the same source code. Buildozer automatically downloads and installs the required programs and create a portable package. This process is configured by one specification file for all supported platforms [3].

6. It is an interface that allows to run code written in C and C++.

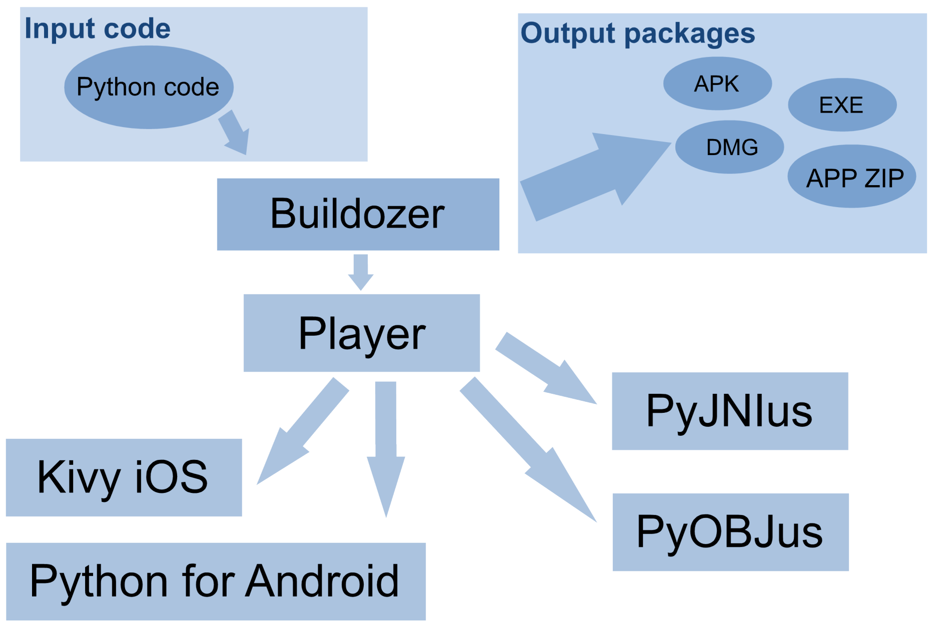


Figure 2.1: Kivy Sister Projects

Kivy Designer This tool enable to design the graphical user interface (GUI) using the WYSIWIG[7] concept. It provides a way for creating a definition of the GUI and customize widgets by KV language 2.3. This designer is written in Python using the Kivy Framework [6].

2.1.2 Community and Support

As mentioned in the research chapter 2, the main development resource is the official Kivy documentation. Besides of that, Kivy provides a way how to solve problems during the development that are not described in documentation. The primary place is a support group. This group is a forum where developers using Kivy ask for help. Questions are answered by the Kivy developers. There is a possibility to get an answer from other members of the community as well, because this forum is open to anybody with a Google account.

7. Acronym for: "What You See Is What You Get". It is a way of editing documents that provides a final appearance interpretation by the text editor itself.

The support group is implemented as a Google Group [4]. The name of the forum is called "Kivy users support" and it already contains around five thousand threads. These questions are in a flat structure. Categorization is possible by tags and there is also a full-text search in all of the topics. The forum contains various topics from basic question to very specific ones.

The Kivy organization does not make any commitment on answering questions. It is possible that nobody answers certain questions. Currently, about one third of all the posted questions are left unanswered.

During the research, the request[8] was made through this forum to verify proposal for Kivy application development. This request was also left unanswered. So, the proposed best practices were based on object oriented concepts and best practices [22, 14] and with Engineering Team Lead from Y Soft Corporation.

Another way to solve development difficulties is to use different forums. The recommended forum is one of the biggest forums on the Internet. This forum is called "Stack Overflow" and in 2012 it had 1.3 million registered users according to the The Empirical Investigation [2]. Currently, there are about 4.6 million of them. This forum contains nearly two thousand questions (based on tag Kivy). These questions are also answered by developers from Kivy organization and also from other registered users. The relevance of questions and answers is possible and verifiable thanks to ratings. Registered users can up-rate and down-rate both questions and answers. This approach does not guarantee the quality of the answer, but it is possible to compare answers for a specific question between each other.

2.2 Development Environment

Setting up the environment is one of the first steps before the actual program coding. This procedure includes choosing the operating system, preparing all the necessary programs like compilers and integrating them into the environment and an optional but usual

8. Link for thread: `https://goo.gl/BZOpGA`

step, installing and configuring an IDE[9].

The first task is to choose an operating system. All main desktop platforms are supported. Potential developers could develop the application on Windows, OS X and Linux. The specific supported versions can be found in the official documentation [7]. As a part of the research, all three platforms were tested. The goal of this comparison is not to pick the best operating system, but to summarize all relevant information that help potential developer to choose.

The use case is to install the necessary minimal set of programs required for creating and running the application and comparing the results. The native installation of operating systems is used for this test to avoid result defacement caused by virtual emulations[10]. Windows 8.1, Linux Ubuntu 15.04 and OS X 10.9.5(Mavericks) are the specific system versions used for test. The application is just a window containing a label with text “Hello word”. The following were selected as the criteria: the availability of the Kivy Framework, availability of the depending programs to run application via a command line and the number and severity of problems during use case execution. The following list describes the generic goals to accomplish the use case, the others are dedicated to specific platforms.

Generic use case

1. Install operating system (not evaluated).

2. Install minimal required set of programs to be able to write specified application and run it using the Kivy Framework.

3. Write code for the specified application and run it via command line.

9. Integrated Development Environment
10. Technology that enables one computer system to virtually simulate another computer system.

Windows 8.1

- Setting Kivy Framework prerequisites
  - Download self-extracting portable package from the Kivy official site [8].
    - Depending on Python version (Python 2.7, or Python 3.4)
    - Depending on Central processing unit (CPU) architecture (32bit system x86, or 64bit system x64)
- Extract downloaded package, this requires about 1.2 GB of space.

  It contains:
    - Python interpreter
    - GStreamer multimedia platform library [5].
    - Minimalist GNU for Windows MinGW[11]
    - Cross-platform development library Simple direct media layer (SDL2)[12].
    - Set of batch[13] scripts running Python and Kivy modules in isolated environment.
- Writing code and running application
  - Default text editor (Notepad) is sufficient to write a basic application.
  - Kivy accepts source files with all types of end of line characters.
  - Run the application by a prepared batch script `kivy.bat` from portable package. It is also possible to include the command into the command line by adding it into the windows environment variable called `PATH`.

11. Development environment that provides access to the functionality of the Microsoft C runtime and some language-specific runtimes: http://www.mingw.org/.
12. Library providing low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D: https://www.libsdl.org/.
13. Script files ("bat" file extension) that contain series of commands that can be executed by the Windows command line interpreter.

Linux Ubuntu 15.04 (Vivid Vervet)

- Setting Kivy Framework prerequisites
  - Install as a package using Advanced Packaging Tool (Apt), which is a standard package manager for Linux Ubuntu.
  - Other supported distributions of Linux have equivalent package compatible with particular package manager.
- Writing code and running application
  - Default text editor (Notepad) is sufficient to write a basic application.
  - Kivy accepts source files with all types of end of line characters.
  - It is possible to create a symbolic link between Kivy and source code folder and then run application as a classic python script. This approach requires adding `#!/usr/bin/kivy` line at the first line into the bootstrap file `main.py`.
  - Run an application by `kivy` command installed and accessible from terminal or run as a Python script using previous step.

OS X 10.9.5(Mavericks)

- Setting Kivy Framework prerequisites
  - Download a Mac OS X disk image (dmg)[14] file and install it by drag and drop into the Applications folder. Also contains application that creates links and sets environment paths to make the Kivy accessible from the command line.
- Writing code and running application
  - Default text editor (Text Edit) is sufficient to a write basic application.
  - Kivy accepts all types of new line characters in default text editor.

14. Disk image file used for program distributions in OS X.

  - Run an application by `kivy` command installed and accessible from terminal.

The comparison did not reveal any serious issue. It is possible to set up environment on any of the tested operating system. There are two main differences between these three platforms.

The first one is installation of external modules. Windows does not require any additional installations. All external modules are included in the distributed package. That is why the package has more than one gigabyte. In Linux, it is possible to use apt, or any distribution specific package manager to install the external modules. These modules are installed into the global environment as is Kivy in the default configuration. It is possible to use pip[15] and install Kivy and external modules into the virtual environment to not influence the whole system. Same approach is used in OS X by default. So all external modules have to be installed using pip.

The second difference is not important in the tested use case, but it makes a significant difference in the real application development. It is a command line access to Kivy. The only way how to run Kivy on Windows is to use a prepared batch file that set all environment paths for current session. This session uses programs from installed package including the python itself. Linux and OS X is able to include Kivy command into the terminal or into the virtual environment. This approach is compatible with IDE configuration and it enables to use standard configuration of these tools to run application from them.

To create a complete set of packages runnable on every supported platform, it is necessary to install all supported desktop operating systems and additional utilities for packaging as is described in section 3.4. Previous description was focused on development environment setting.

2.2.1 IDE Configuration

Development environment preparation research discovered problematic topic in application development. It is necessary to configure IDE to be able to interpret, run and debug the application.

15. Tool for installing Python packages.

In most cases it is required to have a compiler or interpret commands accessible from command line. Setting up an IDE to use Kivy generally involves pointing it to the appropriate interpreter and possibly configuring environment variables. Kivy describes how to do that for three IDEs: PyCharm, Visual Studio and Eclipse in the wiki section on the "Setting Up Kivy with various popular IDE's page"[16].

Windows Windows is, in comparison with the other operating systems, the most difficult to configure the IDE in. Classical development environments are not made to set python as interpreter for batch files, the same way that Kivy uses for running on Windows. Research was made on Visual Studio Enterprise 2015, Eclipse version Luna Service Release 2 (4.4.2), PyCharm 4.0.4. All of them allow to set batch file as an interpreter.

To run the application in Visual Studio, it is necessary to download the Python tools for Visual Studio feature from Programing Languages section. Another step is to create an interpreter using `kivy.bat` file instead of the classic distribution of Python. This interpreter environment can be created in the `Tools/Options` menu. Configuration settings are in the `Python Options/Environment Options` submenu. Then it is required to set following Environments Settings:

- Path: `<KIVY>\kivy-2.7.bat`
- Window Path: `<KIVY>\kivywineenv-2.7.sh`
- Library Path: `<KIVY>\Python27\Lib`
- Architecture: Depends on Windows architecture (x86/x64)
- Language version: 2.7
- Path Environment Variable: PYTHONPATH

After this configuration it is possible to develop the application right into the Kivy. It is required to set Python Environment in Solution Explorer to the newly created Python Environment. All features of Python tools are available for development i.e. debugging. There

16. Ide configuration maunal `http://git.io/v01tm`.

is just one disadvantage. Editing of KV files that contain GUI definition 2.3.1 does not have syntax highlighting. It is also unavailable as an extension that would cover this feature's absence.

Eclipse is configured in a manner similar to that of Visual Studio. To run Kivy application, it is required to install the PyDev extension to be able run classic Python applications. The next step is setting up the Python interpreter. The configuration is more simple than in Visual Studio. In `Window/Preferences` tab menu action in `PyDev/Interpreters/Python Interpreter` sub section, it is required to create a new interpreter and set Interpreter Executable path to `kivy.bat` batch file. Then in `Run/Run Configurations` tab menu item open context menu on `Python Run/New configuration` and create new configuration. The last step is to set configuration Project to point to Kivy application, point Main module to `main.py` file and on Interpreter tab select newly created interpreter. Now, it is possible run or debug applications using Kivy Framework as an application written in pure Python. The KV language syntax is also absent for the syntax highlighter and no extension for this functionality is available as of today.

The last tested IDE was PyCharm. Configuration is nearly the same as in Eclipse. In `File/Settings` tab menu item in `Project/Project Interpreter`, it is necessary to create a new local interpreter that points to `kivy.bat` file. Kivy interpreter is the set as an external library. It is listed in Project file tree under the External Libraries node. This IDE has one big advantage in comparison with two previous ones. It is designed specifically for Python development. It is not necessary to install any extensions. It is also possible to import syntax highlighter for the KV language. This editor allows to use intellisense, syntax highlighting and debugging in python code, because KV language syntax allows to use Python code directly in the KV syntax.

Linux and OS X These two operating systems have very similar settings. Visual Studio is not supported because it is impossible to run this IDE on these platforms. The tested version is strictly available only on Windows. Eclipse has exactly the same configuration, as is described in previous text. The only difference is that as a path to interpreter, the path to Kivy in the `bin` folder is used. The problem with syntax highlighting for KV files is the same as on Windows.

PyCharm configuration for Linux is exactly the same as on Windows. It is different in case of OS X, though. It is necessary to perform one additional step before development. That is the configuration of the interpreter environment variables. This configuration can be set in `Run/Edit Configurations/Environment Variables` in the created interpreter in environment section, there is a property called Environment variable. It is necessary to set these variables as it is stated on the official Kivy manual referenced in the beginning of subsection 2.2.1

2.2.2 Raspbery Pi

Kivy also offers a support for Raspberry Pi. This platform was not included, but there is a prepared distribution for Raspberry [9] with a manual and required resources about this approach.

2.3 Application Development using Kivy

This section analyzes the base architecture of applications written in Python using the Kivy Framework. It focuses on general building blocks and components of the framework that are used in developed application. The main task of the following text is to provide general knowledge on how Kivy works. This information will be used in the implementation part of the thesis 3. All the following information is based on the official Kivy documentation section Programming guide [7].

The basic concept of an application is that the whole user interface is one single canvas. This canvas is loaded while the application is started and it is reanimated using a clock module. This means the user interface is periodically refreshed based on the application's current state. The content of the canvas is defined by widgets. Widgets on a canvas are organized in a tree.

The application live cycle is similar as in Android native development, which is described in thesis about Android development [12]. It is possible to save the state before the application is closed or

```
import kivy
kivy.require('1.9.0')

from kivy.app import App
from kivy.uix.label import Label


class MyApp(App):

   def build(self):
      return Label(text='Hello world')

if __name__ == '__main__':
   MyApp().run()
```

Figure 2.2: Kivy Simple application example

when it is suspended and allows brining the application to the front by the system if requested. This live cycle is described in appendix B.2. Creating a Kivy application structure comprises of the following steps. The first one is creating a bootstrap file called `main.py`. Then, it is necessary to create a class that inherits from the `kivy.app.App` class and implement its `build()` method. This method is responsible for creating a tree of widgets. The tree is described in the following text. For the first test, it is possible to return an instance of `kivy.uix.widget.Widget`. That class represents a base class for most of the widgets. The last step is to call the `Run()` method on the new instance of the class inherited from `kivy.app.App` in `__main__()` method. This configuration creates a new application that contains only one empty widget (an empty window).

Architecture is divided into the three layers. The lowest one uses the open source projects that provide a way how to call native API provided by operating systems 2.1.1. Above this layer there are three components, Core, Graphics and Inputs. The core block contains a set of providers for managing text, audio, video, images etc. The graphics block is used for editing the canvas as a drawing. The last is the Inputs block. It processes motion events and distributes them to the system. Above these two layers is a top level layer. This layer is divided into the two sub-layers. The first layer encapsulates the functionality of widgets in the widget tree and their visible definition described by the KV language and the second sub-layer ensures connection between widgets tree and the second layer by modules like Cache, Clock, Gesture, Event loop and properties. Whole architecture is summarized in figure B.1.

2.3.1 Python and KV language

Kivy respects the concept of dividing presentation and business layer. For this purpose, it is possible to use the KV language for defining the user interface and classic Python code for business logic implementation. This concept makes implementation more maintainable and easy to read for external developers.

The KV language syntax is a mixture of xml, json and Python syntax rules. An example of syntax is in figure 2.3 which allows to create complex UIs, defining widgets and their properties and setting a widget tree structure. The structure is defined by indentation like in Python. Properties consists of name-value, divided by double dots. Name as a classic string and the value is pure Python code. This approach makes definition easy to read with all the advantages of Python syntax. GUI definition is based on templates. It is possible create a widget and inherit business logic, including properties, by inheriting build-in widgets. Another feature is to organize GUI in layouts.

KV language source files have the `*.kv` extension. Naming convention defines that source file with Python representation containing business logic and the GUI definition kv file have the same name. If a developer wants to use a single responsibility principle and have one Python class in one file, it is necessary to specify all kv files that are used in UI in the `build()` method. Loading of these files is provided by the `kivy.lang.Builder` class. The method used for loading kv files is called `load_file(path)`.

There are two ways how to create a GUI definition. The first one is by a text editor. The only editor that offers syntax highlighting is PyCharm 2.2.1. With an extension it is possible to create complex UIs in this IDE using syntax highlighting and intelliSense. The se-

```
<StyledButton@Button>:
   text_size: self.size
   font_size: '25sp'
   markup: True

<Controller>:
   id: my_custom_label
   info: str(self)
   BoxLayout:
      orientation: 'vertical'
      padding: 20
      StyledButton:
         text: 'My controller info is: ' + root.info
         on_release: root.do_action()
      Label:
         id: my_custom_label
         text: 'My label before button press'
```

Figure 2.3: KV Language Sample

cond approach is to create a definition by the Kivy Designer 2.1.1. This program is specialized for creating GUI definition files using the WYSIWIG approach. This designer is also written in Kivy.

2.3.2 Widgets and Behaviors

Widgets are organized in a tree. That means one node could have zero, one or multiple children in every single level. The tree has only one root node and it contains two types of nodes. The first ones are functional widgets and the second are layouts. Layouts are responsible for organizing the position of their children. Widgets hold the functionality and functional properties.

When the tree is loading (in `build()` method), Kivy starts to load the root widget and if it has children, it recursively load them until the leaves of the tree are reached.

In general, all widgets and layouts inherit from the base class `kivy.uix.widget.Widget`. This class provides a basic API for event dispatching, passing through the tree and working with the canvas that the widget contains. It is basically the most general building block that is used for an application construction.

Layouts are like managers that define where will the child widgets be placed, and what size will they have. The base size of a widget is inherited from the parent. In case of more child widgets, the sizes are divided equally. It is possible to define the size ratio relatively between sibling widgets or by an absolute value.

Kivy Framework provides a means of setting the concrete size of the widget for the specific platform, display resolution or display density. In combination with layouts, it is possible to create a complex GUI that is responsive and able to be used on various devices.

Layouts This type of organizational widgets enables you to create a complex user interface that is independent on device and its attributes. The base set contains eight layout widgets that allows to create all types of user interface by nesting approach. Relative layout provides an API for positioning and setting the size of child widget. Scatter layout has the same functionality, but it offers more flexibility in translating, scaling and rotating field. For organizing widgets in a horizontal or a vertical line, it is possible to use a box layout. It is also easy to use a stack layout to organize widget blocks in rows or columns without additional ensuring, if the next widget should fit to the remaining place or it should be placed on the next line. Combination of a box and stack layout is grid layout that enables to align child widgets in a grid by setting column or row count. Anchor layout is used for centering child widget to the center or on site of the layout canvas. The last one is the float layout which has no restriction for position and a size of the child widget that attributes has to be set by properties.

Widgets The base widget is a label. This widget provides a string property “text”. Value of this property is rendered in GUI. It is also possible to use complete API of the “widget” class. On this widget, there is a button and a text input built in by default. In combination with another form widget, such as check box, radio buttons, drop-down list etc. It is possible to create an interface for user input and interaction.

The other group is a manager set. This group mainly contains a screen manager and carousel. These two widgets allow for changing its content. The content itself is made by child widgets. An animation is displayed when changing the content. It is then possible to have animated transition between different content out of the box.

With this set of widgets, it is possible to create a complex UI that contains semantically different content and is able to interact with user and all in responsive and flexible layout.

Behaviors Kivy provides many other extended widgets besides the base ones. This structure is achieved by using the widget inheritance. Python offers multiple inheritance. That means a widget can use the functionality of more than one parent widget.

To make a system in inheritance, Kivy uses behaviors. The idea of behaviors is to encapsulate properties and events into one object and make them reusable in other widgets. That means it is possible to replace the standard widgets with custom ones by implementing the correct behavior. For example, it is possible to create a new widget with two base classes. The image widget and the button behavior. The result of this combination is a widget that appears like an image but provides all the events as a button class without any appearance aspects of button.

2.3.3 Properties and Events

Properties and events are made to a persisting state of widget objects and hold implementations of a reaction for user interaction, this reaction is called the Callback action. A property represents an extension of the classic class properties in Python. Base property classes are implemented in the `kivy.properties` module. This group of properties provides a functionality for value checking and value validation, ability to bind and observe and to be able to react on property's value change and better memory management for the property live cycle.

Events are actions that are processed after the property of a widget is changed. It is possible to use that to process user interaction and to react to property change. Events bubble up from the most recently added widget through the widgets in widget tree to the leaves and then backwards. It is possible to reverse this order by calling base implementation of a currently called callback on the parent class. Another option is to process an event and kill the event in the current callback method by returning the `True` value. By combination of these approaches, it is possible to distribute events to all the nodes in the widget tree and also to save the processor time by killing the event after successful processing.

Creating a Kivy property automatically creates one event that is fired every time the value is changed. The naming convention for this event is `on_<PROPERTYNAME>()`. After that, it is possible to bring a custom callback on this event or overwrite the implementation of this event.

2.3.4 Clock and Animations

User interaction with an application is processed by the main loop. This loop is managed by a clock. Every clock tick is processing coming events from the GUI, network, another threads etc. The complete set of operations in the main thread is described in diagram from Kivy official documentation copied in appendix B.3.

It is also possible to create animations besides the user events processing. It is possible to plan to animate the widget every tick of the clock. It is also possible to use predefined animations to create a smooth modern application with interactive GUI. Kivy Framework also offers predefined transitions and it is also possible to define custom ones by the provided API.

The clock provides an API for planning to schedule and callback action once or to schedule it periodically. This approach is similar to JavaScript [17] and this exact functionality is used in animating widgets.

17. Programing language used to make the web pages interactive and providing a way of animating HTML elements by scheduling animation actions `https://www.javascript.com/`.

```
class MyClass(Widget):
    presure = BoundedNumericProperty(1, min=0, max=100)
    store_item_manager = ObjectProperty(null)
```

Figure 2.4: Kivy Properties Declaration

2.3.5 Development Guidelines

These topics are not described in the Kivy official documentation. But it is necessary to respect a set of rules to make the application maintainable, easy to read and to make for an easier orientation in its code. This section focuses on object programing paradigm and modern design patterns taught, for example, in the Python course PV248 [14] and in the book "Concepts of Programming languages" [22].

The first rule is to follow some coding style guide. Official style guide described on the official Python site [18] is PEP 8. This guide describes how to format the code, how to document it and how to use that format style to orient yourself in the code. With this set of rules, it is possible to understand code written by someone else effectively.

The next approach that make code more maintainable is the concept of responsibility. That concept states that every class should have responsibility over a single part of the functionality provided by the software. That means one class file and one kv file for one class code with a GUI definition. It is also good to name that file the same as the class name contained in the file, just in lower case.

To create the code of the widgets readable it is useful to declare Kivy Properties according to the Kivy official documentation. This approach allows you to see the list of settable properties right from class definition of the widget with default values without the necessity for comments. On the figure 2.4 it is visible that it is not necessary to define the default value in a constructor. Kivy also automatically binds the properties with possibility to validate or make other processing when the value of the property is changed. Another advantage of Kivy properties is the possibility to define the value in KV language.

To prepare applications for using different data layer or a different type of presentation layer, it is good to divide the program into three layers. The presentation, business and data layer. Data layer is responsible for loading and storing the data. It generally holds the persistence of the application. Business layer is responsible for using the data layer and providing it to the presentation layer. The presentation layer is responsible for rendering the data in GUI and dispatching user interaction into the business layer.

Code Structure The main goal of this section is to create a structured and maintainable application. This goal could be achieved by dividing the application into layers and design classes and GUI definitions according to object paradigm as was mentioned in the previous subsection 2.3.5.

Application structure should respect the three layered architecture. The most general way how to achieve this is to create three modules. The module is in the Python folder with an initialization script, which is a file called `__init__.py`. Optionally, it is possible to create a separate folder for resources such as images, localization files etc. Under the layer modules, it is possible to nest another semantic hierarchy based on semantics.

To give a better idea on how to create an easy to read source code and maintainable application, it is possible to start with the following folder structure 2.5. That example is just a template of how is possible to create a structured application. The official Kivy documentation does not provide best practices on how to structure the code. This sample is to give you a general idea. In fact, the naming conventions of folders, according to PEP 8, are wrong because of better readability. Name of the folders should be in lowercase if you were to follow the PEP 8 rules.

2.3.6 Localization

To create a multicultural application, it is necessary to design the GUI to be able to handle different length of displayed strings. Beside that, it is necessary to create a robust user interface that is able to handle all culture specifications as special characters in alpha-

```
Sample application
|-- Presentation Layer
|   |-- Initial Screen
|   |-- Menu Screen
|   |-- Setting Screen
|   `-- Main Screen
|       |-- Menu
|       `-- Content
|-- Business Logic
|   |-- Settings Provider
|   `-- Data Provider
|-- Data Layer
|   |-- Rest Service
|   `-- Database Service
`-- Resources
    |-- Images
    `-- Resource Strings
```

Figure 2.5: Sample Folder Structure

bet, RTL and LRT layouts etc. Kivy does not provide an embedded functionality to create multicultural GUI.

It is possible to implement this feature, but all of the functionality mentioned in the previous paragraph has to be implemented from scratch. It is possible to create text widgets from the Kivy default set by inheritance and implementing them the required functionality. To make these widgets able to display all special characters, it is possible to use encoding and custom font. To ensure that these widgets show text in the defined language, it is possible to create a Python class (language manager) that returns string according to a language and string key. Then, in the Kivy KV language GUI definition, import this language manager and use it for setting text property of text widgets. Custom font could be also used for rendering font icons. The font uses vector graphics. This approach allows for using scalable icons without the necessity to store images in all required sizes.

The most complicated part is to defining the GUI for RTL[18] cultures. Kivy provides widgets that are able to work with input and output text in LTR.L[19] To be able to create a layout and text widgets in RTL, it is necessary to implement the complete functionality. That step requires complete knowledge about rendering, positioning and animation implementation in Kivy Framework. To cover all the functionality that provides Kivy, is it is necessary to add functionality to all widgets including the text and layout ones.

2.4 OS-specific functionality

All of the platform specific functionality is provided by the project Plyer 2.1.1. This fact makes development application using Kivy Framework dependent on Plyer. All unsupported functionality that Plyer does not provide is necessary to implement by developer. This implementation has to be implemented separately for all supported platforms. This approach loses the advantages of the multi-platform development. This approach also increases the amount of configuration for packaging the final distributable application.

2.5 REST, json and hal

Representational state transfer (REST) is an approach for creating, reading, updating and deleting information on servers using the hypertext transfer protocol (HTTP) or its secure version (HTTPS). It is an interface architecture. The information that are transferred between requester and server are serialized into XML[20] or JSON[21].

Terminal server REST API provides an ability for managing print jobs and dependent objects. The information is represented in JSON formatting. The data is divided into the two main groups. The first one is the data part, which contains the actual information about

18. Writing system where text starts on the right and continues to the left. This system is used in Arabic speaking countries.
19. Writing system that starts writing on the left and continues to the right.
20. Extensible markup language that is used for structuring information in text.
21. Javascript Object Notation is definition of how the object in JavaScript can be defined by text.

the objects. The second one is a link part. It is provided by hypertext application language (HAL[22])

Example of the simple object that is retrievable from a server as a request body is shown in figure 2.6. It represents user information. The `_links` member objects contain a list of links that are possible to access.

22. Hypertext Application Language defines hyperlinks between resources http://stateless.co/hal_specification.html

```
{
   "name": "John Doe",
   "balance": "150 EUR",
   "billingCode": {
      "code": "5.1",
      "description": "Internal"
   },
   "hasEntitlement": true,
   "userRights": [
      "copyBw",
      "copyColor",
      "scan",
      "printBw",
      "printColor"
   ],
   "logoutTimeout": 15,
   "_links": {
      "self": {
         "href":
            "et/v1/100000000027/auth/loggedUserInfo"
      },
      "printApplication": {
         "href": "/et/v1/1000000000000027/jobs"
      },
      "scanApplication": null,
      "copyApplication": null,
      "billingCodesApplication": {
        "href": "/et/v1/1000000000000027/billingCodes"
      }
   }
}
```

Figure 2.6: JSON and HAL example

Chapter 3

Implementation

The output of this chapter is a set of development guidelines for potential programmers, installable packages for all supported mobile and desktop platforms and a list of obstacles blocking their creation. Focus is put on structured description of problematic topics of development in Kivy. Tips and guidelines are made by modern principles of object oriented programming, such as dividing the presentation layer and the business logic or the Single Responsibility Concept. These principles were taken from following the sources [22, 14].

The goal is to create functional multi-platform client for Terminal server REST API. This API provides set of functions for the management of printer actions. These actions are called jobs. Job is a printable file sent into the printer with meta data used by the printer. The final application connects to one specific printer. The specification consists of a protocol, domain, port and path to accessing the printer by the HTTP protocol. User logs in the job management interface. After login, the user is able to manage jobs sent by himself into the connected printer. This application is in general a client that is able to visualize print jobs and provide user friendly interface to manage these jobs. General workflow is shown on diagram 3.1.

The minimal part of the functionality the application provides includes ability to log in into the specified printer using a PIN. Another part is managing print jobs, which includes previewing, sending to print, marking as favorites. The last part is to preview the list of billing codes and list help topics that are able to display manual.

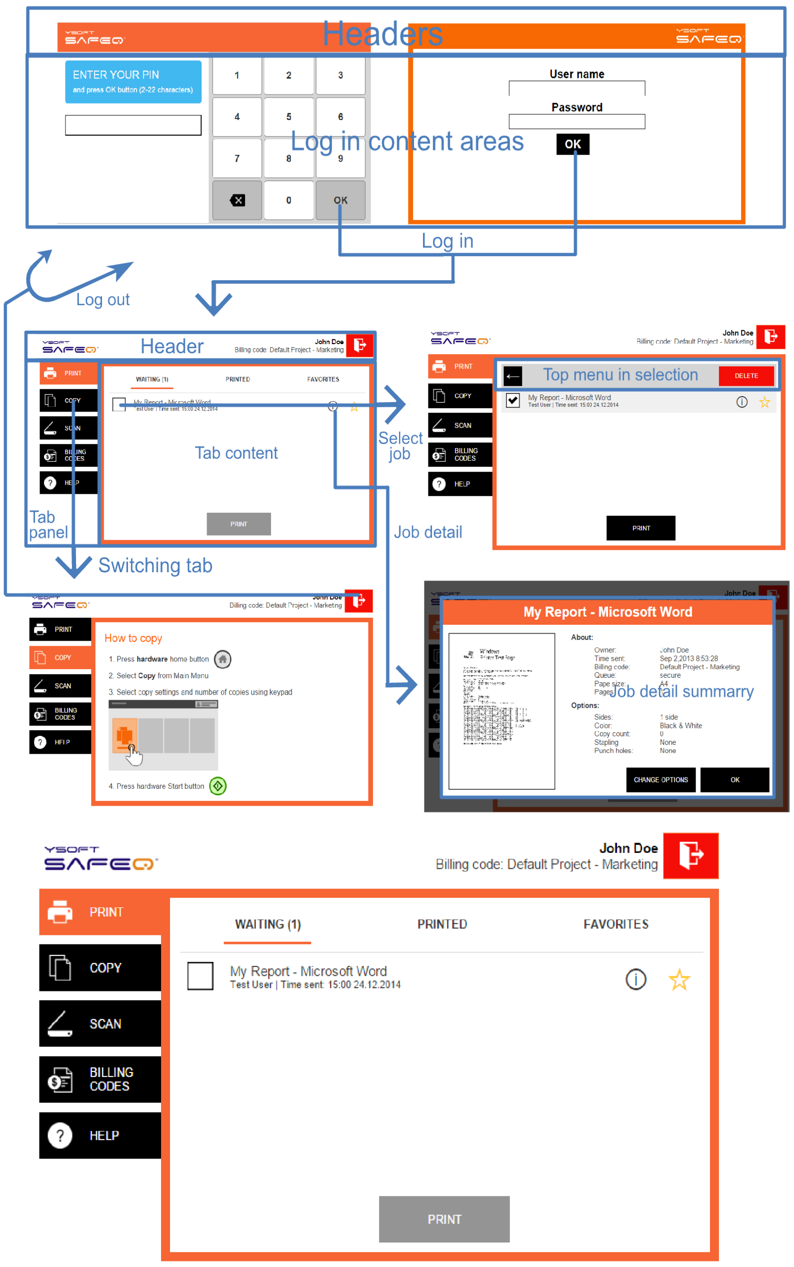


Figure 3.1: Mockup describing general workflow in developed application.

3.1 User interace

The user interface was provided by Y Soft in four iterations. All iterations are available in attachments A.

Final design is described in figure 3.1. This diagram will be referenced from the following paragraph.

The application is divided into two main views. The first view is used to provide login. The second one provides print job management. These two will be called screens. User is able to log in by PIN and also by user name and password. Login screen is displayed in the first two images of the mockup. After a successful login, the user is shown a second screen offering the main functionality. This screen is divided into the three main zones. The top zone is used to display information about the logged user, left zone provides a set of tabs that change the content of the third zone. Detailed analysis is provided in subsection 3.1.1.

3.1.1 Defining GUI

Login screen provides an ability to log in via a PIN and by user name and password. It is necessary to display two separate user interfaces for each type of login. In first two images in figure 3.1, it is shown that the screen is split into the two main areas. The first one is the static top header. The logo of the Y Soft is placed on the right site and the color theme is unified with the main screen of the final application summarized in figure 3.1. The second zone contains form for login. The PIN part displays the label and an embedded keyboard. This approach is possible to use on devices without keyboard i.e., printers. The second approach describes how to make form without an embedded keyboard and rely on its resource on a specific device.

The second and main screen contains a more complex GUI in comparison with the login screen. It is possible to divide the main screen into the separate zones with dynamic content and the static header. The header contains the same logo as the login screen, but it is placed on the left site. Right side of the top header is designed to hold information about the current user. The rest of the screen is divided into the two main zones. On the left side, there is a placeholder for tabs. In this zone, there are five tabs: Print tab, Copy tab, Scan tab.

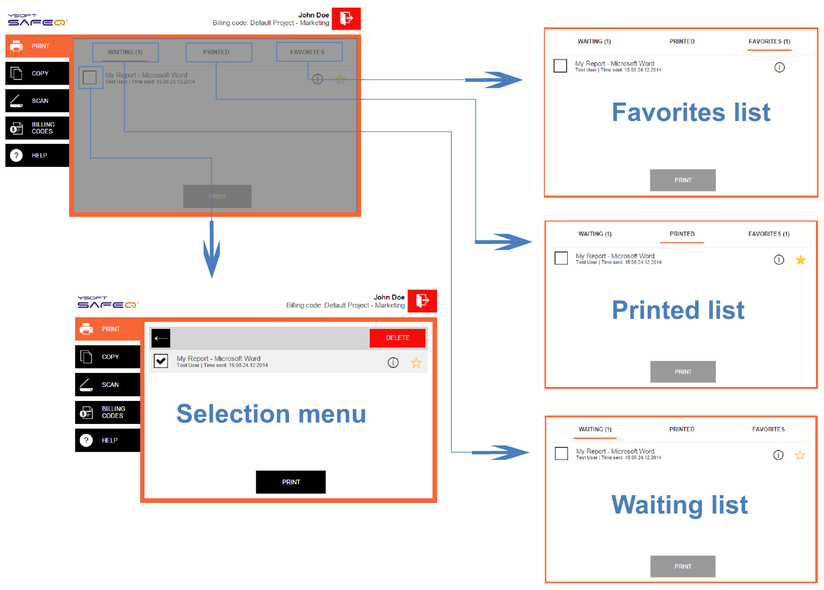


Figure 3.2: Print Tab Mockup

Billing codes tab and a Help tab. These tabs are used for switching the content of the zones on the right zone from tab buttons. A more detailed description is in paragraph 3.1.2. Possible views are shown in figures 3.2 and 3.3.

3.1.2 Components

According to the Kivy official documentation [7], it is recommended to use the `ScreenManager` widget to divide the application into separate screens. In this specific case, it is the login screen and the main screen. It is possible to nest these widgets to manage dynamic content more granularly.

According to the figure 3.3 , the Root Manager responsible for switching content between the login screen and the main screen. The Content Manager tab controls content switching according to the selected tab on the left site. When the Print tab is selected, the Content Manager sub tab displays a list of jobs according to the

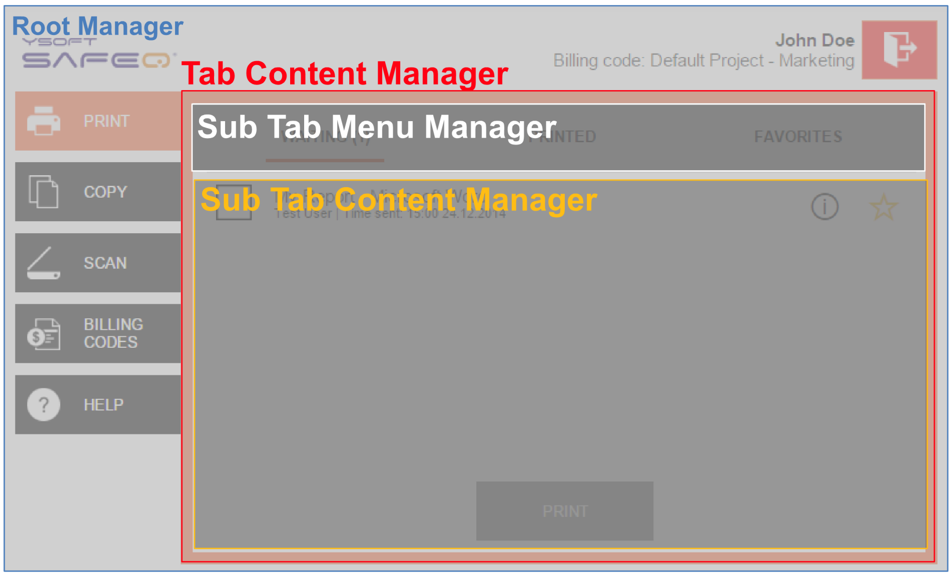


Figure 3.3: Screen Manager Division

selected sub tab. These sub tabs are shown under the Sub Tab Menu Manager. If one or more items in the print job list is selected, Sub Tab Menu Manager changes its content from sub tab menu into the selection menu. This change is visible in figure 3.2.

Login screen

Authentication screen contains two components. First one is a basic form containing two text inputs, labels with one button for submitting the button. In the second component is one text input and a group of buttons, simulating the numeric keyboard. All of the following components are already implemented in the framework. The only additional code is for event handling and styling the widgets. The static part only contains a logo.

Main screen

The second screen contains complex information architecture. The header contains a logo and one label for the current user name and billing code and one styled button. The left tab panel is a group of styled toggle buttons. The Kivy Framework does not provide as robust an API as is required for the left tabs. So the tabs are styled toggle buttons with a constraint that enables to select exactly one button. Switching the tab content is implemented by Tab Content Manager described in subsection 3.1.2.

Zone managed by Tab Content Manager contains five type of layout. Following description is sorted from the simples to the most complex one. Copy and scan content is only an image describes how to perform action on the printer, because it not possible to perform these actions without presence of the user next to the printer. The help tab is a group of five buttons. These are only styled buttons that open a page with manual site about a specified topic. Billing codes are a list of billing code items. This component is implemented as a list view using dictionary adapters described in paragraph 3.1.2.

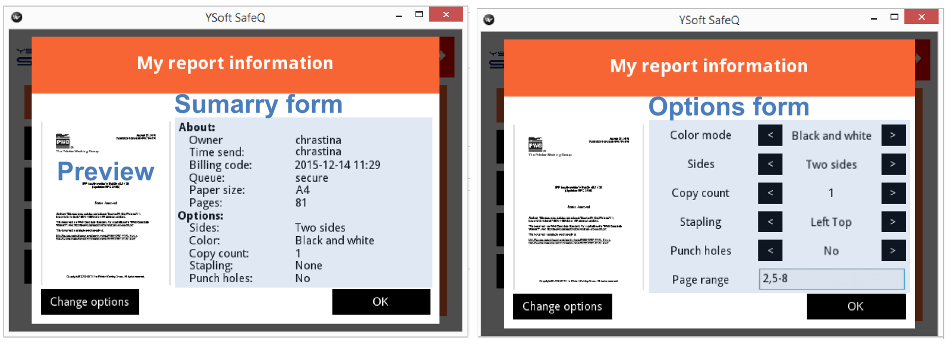
The most complex component managed by the Tab Content Manager is the Print tab. This component contains three lists organized into the three tabs. Tab selectors are implemented as styled toggle buttons, similarly to the tab selectors in the left tab panel. Additional support for animation is also implemented. This feature is described in 3.1.3. Lists contain print job items. The first list is called "Waiting" and it contains jobs that are ready for print. "Printed" list lists already printed and "Favorites" list displays job items that are marked as favorites. Every item in every list has similar layout and provides an information button for showing the modal dialog that is described in paragraph 3.1.2. The last component is the selection menu that is shown, if one or more job items are selected. Selection menu offers actions for selecting or deselecting all items in the list and the delete action. Delete button visibility depends on value of the `_show_delete_button` boolean property. Delete action is visible only in list containing items ready to be printed.

Lists List visualization belongs to the most robust form elements. Lists can theoretically contain an infinite number of items. This requirement is satisfied by the principle of adapter visualized in figure B.4. This concept consists of three main classes.

The first class represents data. Kivy Framework accepts `SimpleListAdapter` that is a wrapper for classic Python list. The second one is a `ListAdapter` that contains more behavior properties. It is possible to specify a selection mode, caching etc. Last type of data adapter is `DictAdapter` which is a wrapper for classic Python dictionary with some additional properties. The second class in the adapter principle is a template. Template is used to display an item in the list. It is based on `SelectableView` and its main purpose is to display data in the user interface. The last class connects the previous two. It is called Argument converter. Its purpose is to transform data item from data adapter to the instance of a template. This concept loads only as much items as necessary to display them in user interface. If any other item is required, it is loaded dynamically in the runtime. This allows to recycle templates without necessity to instantiate them again.

Kivy provides more simple implementations of list visualization that require only list of data items and embedded templates based on a button or on a label. This set is sufficient for primitive data types and simple view requirements.

Specific example of using list visualization is in the billing codes tab and in all sub tabs in the print tab. Billing codes tab is less complicated. As a data adapter, a dictionary of a billing code objects retrieved from the server is used. Argument converter is a lambda expression that gets a billing code id, name and description from the data objects and pass them as a constructor argument of the template. Template `BillingCodeItem` is a class inherited from `SelectableView` and a `BoxLayout` described in paragraph 2.3.2. Template accepts properties from argument converter in the constructor and creates a widget for a list view. The same principle is used in sub tab lists in the print tab. The only difference is that list items allow to select items by a check box. It is also possible to show the modal dialog (description in paragraph 3.1.2) and set or unset the item as favorite. Selection is implemented as a standard event of a check box that calls `select` method of `SelectableView`. Favorites functionality is implemented as a styled toggle button that sends REST request with specific id of the list item on server.



About:

Owner	chrastina
Time send:	chrastina
Billing code:	2015-12-14 11:29
Queue:	secure
Paper size:	A4
Pages:	81

Options:

Sides:	Two sides
Color:	Black and white
Copy count:	1
Stapling:	None
Punch holes:	No

Color mode	< Black and white >
Sides	< Two sides >
Copy count	< 1 >
Stapling	< Left Top >
Punch holes	< No >
Page range	2,5-8

Figure 3.4: Job Detail Modal Dialog

Modal dialog Modal dialog is implemented by a widget `Popup`. This widget can contain any type of child widgets. It is basically another root of the widget tree injected into the classic widget tree when the method `open()` is called. Passing data to the modal dialog is the same as for other widgets, by Kivy properties or as constructor arguments. Closing the dialog is implemented by `dismiss()` method.

Specific example of the modal dialog containing complex GUI including another screen manager, form and carousel is implemented in every list item in the print tab content. All the logic is implemented in `JobDetailPopup` widget. Every print job list item holds the id of the job that it represents. This id is passed into the modal dialog, dialog sends request for getting the detail of the job and renders the summary form and the editable form using carousels. Screen shot containing job detail modal dialog is in figure 3.4.

Layouts Layout represents the concept of widget placement the directive way. Every type of layout described in paragraph 2.3.2 has its own implementation of positioning and sizing the child widget. Widget size could be set by ratio according to the size of the layout widget or by fixed size in one of Kivy units. Kivy property `size_hint` is used for a set ratio that is a list of `size_hint_x` and `size_hint_y` that specifies ratio of width and height.

A specific layout usage with a login screen containing anchor and box layout and a help tab content of the main screen was cho-

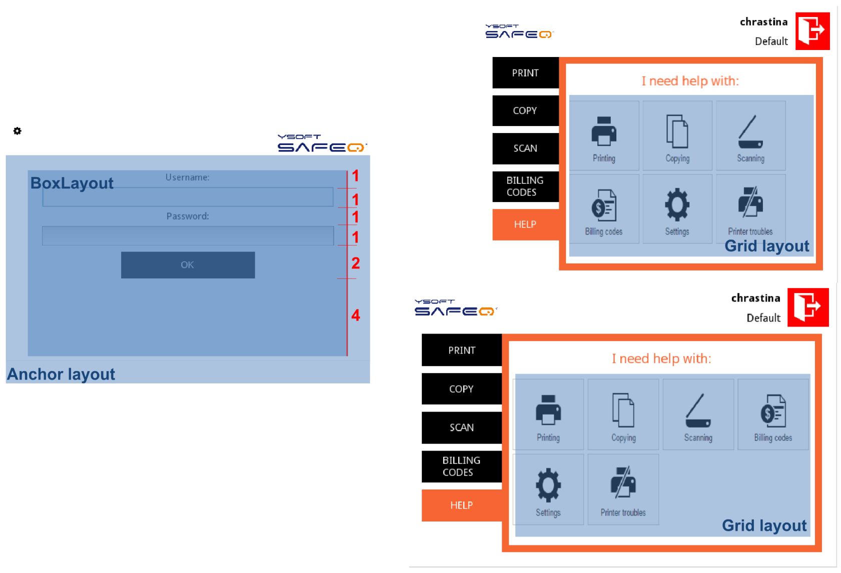


Figure 3.5: Layouts Example

sen as an example of the grid layout.

Example interface is highlighted in figure 3.5. On the left site is a login screen. It contains an anchor layout to achieve padding and centering of the child. Child of the anchor layout is box layout containing form elements as labels, text inputs, button and empty space. Empty space is used as a place holder to display embedded keyboard, because Kivy Framework does not adapt user interface, when the keyboard is shown. There is also a highlighted y-axis ratio of every single form element by red color. On the right site a usage of grid layout is illustrated. By the width of the grid layout, the number of help buttons that fit in one row is calculated. Layout is recalculated for every resize. This ensures responsive behavior as is visible on the screen shots.

3.1.3 Animations

Animations are one of the features how to make an application more user friendly. Feeling from the application with animations and loaders affect looks more smooth than without it.

Animations are embedded into the screen manager and carousel that was described in 2.3.2. For carousel, it is possible to define length of animation of switching slide. For application a default duration 0.3 seconds between slides was used. Screen manager provides an API to define transitions between slides. It is possible to choose one from embedded transitions or to implement a custom one. For implemented application, slide transition with different transition directions was used. For example the Content Manager tab from figure 3.3 sets the direction according to the direction between currently selected tab and newly selected one. This make an impression that tab contents stand on each other.

An example of the classic animation is in sub tab menu in print tab content shown in figure 3.2. Under the tabs in sub tab selector menu, there is an orange rectangle that identifies the selected tab. Detail of change is shown on figure 3.6. That animation is implemented as a separate widget under the tab selectors. This widget is a placeholder for the orange rectangle. When the selected tab is changed, an animation is created and the rectangle changes position by linear speed under the selected tab selector. Moving the rectangle is implemented by changing x-axis coordinate. The duration of the animation has to be set for the same duration as transition of screen manager that controls sub tab content to synchronize tab animation with content switching.

Other implementation method for animation is to use clock object, described in section 2.3.4. This approach is used for loader. This loader is used in all cases until an information is loading, for example when job items are loaded. It is a font icon from Font Awesome[1]. Using custom fonts was described in subsection 2.3.6. Every half second is fired an animation that changes angle of the label containing loader icon. This ensures that icon is rotating.

3.2 Communication with Server

Data layer of the application is provided by Terminal Server REST API. The application itself does not store any data. The server is able to communicate with various devices. The API is able to manage

1. Font containing scalable vector icons. Home page: http://git.io/vGTdX.

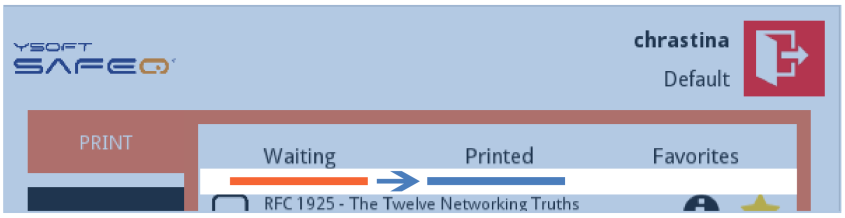


Figure 3.6: Sub Tab Selectors Animation

Get authentication methods.	AUTHENTICATION
Log in using user name.	
Get information about currently logged user.	
Log user out.	
Get jobs waiting for print.	JOBS
Get already printed jobs.	
Get jobs marked as favorites.	
Print one or more jobs.	
Delete one or more jobs.	
Mark job as favorite.	
Unmark job as favorite.	
Update job information.	
Get billing codes.	BILLING CODES

Table 3.1: Required REST Operations

more objects than an implemented application requires. The application requires the API for authentication, print job management and for billing codes. The rest of the API is not used and not described. The required set of operations is summarized in figure 3.1.

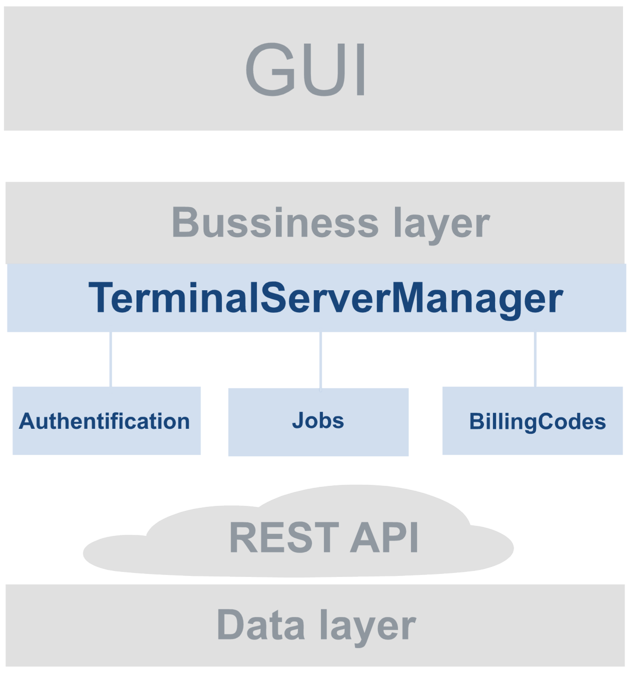


Figure 3.7: Data Access Layer Architecture

The required set of API operations can be semantically divided into three groups according to the subject. According to object responsibility concept, the communication is divided into three classes called `Authentication`, `BillingCodes` and `Jobs`. These three managers are called through one data access manager called `TerminalServerManager` visualized on figure 3.7. This approach allows to extend data layer by creating a new class and connect it into the data access manager. It is also possible to switch data layer effectively.

3.3 Development guidelines

The following section is dedicated to specific guidelines that are used for the proof of concept application. These guidelines were made during the development process. The goal is to create a set of rules to make an application extendable, maintainable and easy to read. All of the rules respect object oriented paradigm and design patterns from these resources [22, 14]. Some of these rules are broken intentionally. These are used as an examples subsection 3.3.2.

3.3.1 Code structure

It is not effective to divide application code structure into the three logic parts to respect the three layer architecture described in paragraph 2.3.5. Data layer on server and its access is divided into the special sub tree in file system and this approach respects the three-layer architecture. Presentation and business layer are in one subtree. That is because separation of business and presentation layer in Kivy is made by dividing widget presentation into KV language files and its logic into python files. It is required to place these files into the same folder and with the same name. This limitation blocks division of presentation and business logic. The only way how to achieve division on file system structure is to use widget python classes just as wrappers for business class objects. This approach was tested on the proof of concept application and it makes the code more complicated and hard to read.

The final folder structure is visualized in figure 3.8. All folders all named by lower case according to PEP 8 specification [18]. Folder `dataaccess` holds classes for communication with server. It contains `contracts` folder, where all required API is defined in one place. The only type of resources used in developed application are images, that is why the resources folder is named `images`. Folder `ui` respects screen managers responsibility described in section 3.1.2. File system folder tree also respects widget tree structure of an application. Kivy Framework does not describe how to structure an application that is why it is created empirically.

3.3.2 Bad and Best Practices

This subsection describes bad and good practices used in implementing the application. This approach was defined during the development. All of the examples including bad practices are in the source code for illustration purposes.

```
SafeQ
|-- dataaccess
|   |-- contracts
|   `-- rest
|       |-- auth
|       |-- billingcodes
|       `-- jobs
|-- images
`-- ui
    |-- loginscreen
    `-- mainscreen
        |-- tablayout
        |   |-- billingtab
        |   |-- copytab
        |   |-- helptab
        |   |-- printtab
        |   |   |-- favoriteslist
        |   |   |-- printedlist
        |   |   |-- topsubmenu
        |   |   `-- waitinglist
        |   `-- scantab
        `-- topheader
```

Figure 3.8: SafeQ Folder Structure

First example is a list of the visualization architecture. How lists work is described in 3.1.2. List is built from two widgets. The first one is `ListView` and the second is list item, which is the widget inherited from `SelectebleView`. It is good to divide an implementation for these two classes into two separate files. Example of this separation is illustrated in `BillingTab` and `BillingCodeItem` classes. These classes are divided and now it is possible to use `BillingCodeItem` in more than one list. Bad practice is to store an item and a list in one file. This example is included in Waiting, Printed and Favorites list on the Print tab. Every list has its own list item definition. All items in all three lists are similar. The only difference is that in the Favorites lists, there is a missing star toggle button for marking job item as a favorite. The best practice is to extract definition for all job items into one item class and implement ability to change visibility of the start toggle button depending on a boolean property. This approach is used in top selection menu described in subsection 3.1.2 that has configurable visibility for delete button.

Another practice is to use KV language inheritance syntax. According to the template concept, described in subsection 2.3.1, it is possible to create widget based on already created one and preset some of its properties, or define new ones. It is possible to include all the code in KV language definition without the necessity to create a Python logic class. Example in code is included in the `loginscreencontent.kv` and `loginscreencontent.py` files. A bad practice is to define widget inheritance link in Python logic. Example of this notation is in the `PinNumberButton` widget. Property values are set in the KV language file and the information that this widget inherits from `Button` widget class in Python logic file. To make the code easier to read, it is possible to define inheritance in KV definition file together with setting widget properties. This is illustrated in the `FormItem` widget. The difference is illustrated on figure 3.9.

3.4 Packaging

One of the biggest advantages of the Kivy Framework is its multiplatform packaging. It is possible to pack the application into one

```
# loginscreencontent.kv
<FormItem@BoxLayout>:
    orientation: 'horizontal'
    label_text: 'default'
    padding: 5,5
    Label:
        id: item_label
        size_hint_y: None
        size: self.texture_size
        padding: 5,5
        text: root.label_text
        color: 0,0,0,1
    TextInput:
        id: item_input
        size_hint_y: None
        size: item_label.size
        padding: 5,5
        multiline: False
```

```
# loginscreencontent.kv
<PinNumberButton>:
    background_normal:
            'images/pinnumberbutton_normal.png'
    background_down:
            'images/pinnumberbutton_down.png'
    color: 0,0,0,1
    font_name: 'DroidSans-Bold'


# loginscreencontent.py
from kivy.uix.button import Button


class PinNumberButtofn(Button):
    pass
```

Figure 3.9: KV language Inheritance Definition

distributable package. This operation consists of compiling for a particular operating system, creating distributable package according to environment and optionally security steps, such as signing, obfuscation etc. Packaging is the most difficult operation for Kivy. The framework has to provide a way how to run an application written in Python on environments with different complier implementations as automatically as possible. Description is included in the 2.1.1 subsection.

Developers from Kivy provide a semiautomatic way for creating packages for all platforms. The packing principle is almost the same for all operating systems. First of all, source codes must be in Python 2.7+, because every packaging tool supports only this version. Support for Python 3.3+ is on the way according to the official documentation, but the specific date is not announced. The next step is to download the packing tools 2.1.1. These tools are platform specific. The next step is to configure compilation and set meta data for environment and final package.

To find out how the whole process works more deeply. As a part of implementation process and one of the goal of this thesis is to create a package for every single supported platform or describe problems that block the creation. In the following paragraphs, the process of creating a portable package for each supported operating system including the packaging problems is described. The following subsection is only general a description of the process, for detailed guide use packaging section in official Kivy documentation [7].

According to section 2.2, it is necessary to install all Windows, Linux and OS X operating systems to create all types of application. The following will be used for this testing: Windows 8.1, Linux Ubuntu 15.04 and OS X 10.9.5(Mavericks).

3.4.1 Windows

Windows uses PyInstaller [15] as a packing tool. It is possible to download PyInstaller using pip. This packaging tool and the Kivy portable package are required to create a package for distribution. When a developer wants to perform any action in the packaging process, he is required to set all necessary environment properties as `PATH`, `PYTHONPATH` etc. These settings can be performed automatically. Simply run script file `kivy.bat`, located in Kivy portable package.

The first step is generation of the specification file. This specification file is generated by running Pyinstaller main script `pyinstaller.py` with parameter `name` and a value represents a name of final package. Another parameter is path to `main.py`, the bootstrap file containing entrance point into the application. It is obligatory to name this file `main.py`. That is the most basic way how to generate a new folder with specification file.This command generates a default specification file with `spec` extension. There are many possibilities how to change the configuration in generated file. All of them are described in PyIstaller documentation [15]. The most important result is a basic template for specification file. This file is displayed in figure 3.10. Specification file has to be edited according to Kivy documentation [7] or PyInstaller manual [15].

The second and the last step is to run application compilation. This is performed by calling `pyinstaller.py` with path to specification file as an argument. The program collects all required resources and compile them into one folder. This folder mostly contains `dll`[2] and `pyd`[3]. It also contains Source code and some metadata.

2. Dynamic linked library - type of container for shared code using in Microsoft technologies.
3. Python script made as DLL.

```
# -*- mode: python -*-
a = Analysis(['..\\Temp\\SafeQ\\main.py'],
             pathex=['C:\\kivy1.8.0-2.7'],
             hiddenimports=[],
             hookspath=None,
             runtime_hooks=None)
pyz = PYZ(a.pure)
exe = EXE(pyz,
          a.scripts,
          exclude_binaries=True,
          name='safeq.exe',
          debug=False,
          strip=None,
          upx=True,
          console=True )
coll = COLLECT(exe,
               a.binaries,
               a.zipfiles,
               a.datas,
               strip=None,
               upx=True,
               name='safeq')
```

Figure 3.10: PyInstaller specification file

The most essential thing in this process is to set all the environment variables as was mentioned above. In case of skipping this step, it is not possible to generate the specification file. It is important to keep in mind that environment path setting is valid only for one session of the command prompt. So the environment set up has to be performed after every start of the command prompt.

Problems

Packaging application for Windows is really vulnerable to errors because of version incompatibility. It is necessary to find in the documentation version history what version of Kivy is compatible with specific versions of PyInstaller. It is also impossible to use an externally installed PyInstaler. The only combination of versions that was able to create distributable package is Kivy version 1.9.0 and PyInstaller 2.1. By following steps for specified version[4], it was not possible to create a functional package. This procedure requires to install PyInstaller into the Kivy portable package using installed pip. Then, it was not possible to import `kivy.uix.selectableview` for unknown reasons. This class was copied into the application sources. After that and another additional action was able to create a portable application.

3.4.2 Linux

Linux distribution is the easiest operating system to run Kivy applications. All requirements for running are provided in one package. It can be installed by `apt-get`[5] or alternative packaging system installed on a speicfic distribution. After the installation, the Kivy Framework can be run as a standard Python program. No packaging is required.

4. Concrete documentation version of Windows packaging manual: http://git.io/vEcfP
5. Package handling utility. http://linux.die.net/man/8/apt-get

Problems

During the installation and running Kivy application on Linux Ubuntu 15.04, there were no problems to run the developed application and any of the examples provided by the Kivy Framework.

3.4.3 OS X

Packaging application for this operating system is similar as on Windows described in previous subsection 3.4.1. One requirement is a portable package containing Kivy Framework. The second prerequisite is also PyInstaller [15]. Generating the specification file is similar, just with another argument `windowed`. Running the compilation is the same. The only difference between OS X and Windows is finalization. Windows package would be distributed as a folder. OS X uses `dmg`[6] files to distribute applications. The standard way to create this image file is `hdiutil`[7] tool. After the compilation, the package contains most of the Kivy modules. It is possible to delete the unused parts to reduce the package size. After this step is program ready for distribution.

3.4.4 Problems

The first tested version of the OS X operation system was 10.11 (El Capitan). During the compilation, PyInstaller writes into the `'/usr'` directory. This version of the OS X contains new System Integrity Protection[8] that blocks any writes into this folder. Deployable package was created on version OS X 10.9.5.

3.4.5 iOS

Packaging process for iPhones and iPads is the most complicated. Prerequisites are summarized in table 3.2

6. Mac OS X Disk Image file.
7. Tools for managing disc images on Mac OS X
8. Security technology used on OS X: https://support.apple.com/en-us/HT204899

Name	Description	URL
autoconf	Package of macros for shell scripts.	http://www.gnu.org/software/autoconf
automake	Tool for generating a compiling script .	https://www.gnu.org/software/automake
libtool	Support script library.	https://www.gnu.org/software/libtool
pkgconfig	Helper tool for setting compiler options.	http://pkgconfig.freedesktop.org
pip	Python packages installation manager.	https://pip.pypa.io/en/stable
cython	Python compiler uses implementation of routines in C language.	http://cython.org

Table 3.2: iOS Prerequisities

The main packing procedure consist of four steps to create runnable application. The first step is to download and compile the Kivy distribution for iOS. Compilation script called `toolchain.py` is included in the distribution package. The next step is to create an Xcode project [23]. This action is also performed by the toolchain script as well as all another steps. After opening the project, it is possible to compile all the external python modules i.e. numpy[9] and include into the project and link to the XCode project. After all these steps, it is possible to build the Xcode project and distribute, same as standard one[10].

Problems

The package for iOS was not created because the required membership to create the portable package is paid[11]. That decision was made with thesis supervisor.

3.4.6 Android

There are two main approaches on how to build applications for Android. The first one is to copy the source code to the Android device and run it by KivyLauncher[12]. This application loads sources from an SD card and provides Kivy runtime environment. Using this approach, it is impossible to publish the application on Android store, however.

The second option is to compile the application locally and get an APK file. Creating an APK can be performed using two approaches. The first one is to use `python-for-android` descibed in paragraph 2.1.1. This process requires Android Software Development Kit (SDK)[13] and Android Native Development Kit (NDK)[14]. After the installation, it is necessary to configure `python-for-android` and

9. Scientific module for Python: http://www.numpy.org/
10. Manual for XCode project distribution: https://goo.gl/SOIerh
11. Developer membership summary: https://goo.gl/ujOoxa
12. Kivy Launcher application on Google Play: https://goo.gl/mrg2So
13. Development kit required to develop Android applications: https://developer.android.com/sdk/index.html.
14. Toolset that allows to use C/C++ for development on Android: https://developer.android.com/tools/sdk/ndk/index.html.

set location of Android SDK and NDK. The last step is to configure APK package meta data and run the compilation process.

All steps described in the previous paragraph can be automated by a tool called `buildozer` described in paragraph 2.1.1. To build an application, it is required to generate a specification file. Edit the configuration file and set the application package meta data, including a list of required system permissions. After that, the rest of the process is performed automatically by one command.

To create the package for Android, it is necessary to use the Linux operating system. There are mainly two possibilities on how to do that. The first one is to use a prepared virtual image with Linux with already installed prerequisites. This approach is easier for installation, but it is hard to deploy application directly to an Android device from a virtual environment. The other choice is to install Linux and all its prerequisites and use native installation. For the testing of package compilation was used a native installation of Linux Ubuntu 15.04 and `buildozer`.

Problems

A recommended step before every single packaging is to update all the required programs. Updating just a single support program can block the compilation process.

The development application uses custom fonts called Font Awesome. File with font definition has to be copied between the standard font to enable the Kivy Framework to load it and include it into the distributable package.

Font definition paths:

- `.buildozer/android/platform/python-for-android/`
  `dist/<PACKAGENAME>/private/`
  `lib/python2.7/site-packages/kivy/data`
- `.buildozer/android/platform/python-for-android/`
  `dist/<PACKAGENAME>/python-install/`
  `lib/python2.7/site-packages/kivy/data/fonts`

Chapter 4

Validation

This section describes the usability issues of the developed application. This chapter is one of the most valuable for deciding whether the Kivy Framework is appropriate for commercial use. The first part is a usability test on the provided use case. In this, a specification of the tested devices and terminal server, initial data on the server, testing use case and results, is described. The rest of this chapter is dedicated to additional testing and deployment on Raspberry Pi.

Use Case Validation

Target Terminal server

Domain with terminal server is `http://demo.ysoft.com`. REST API is listening on port `5021` on path `/et/v1/1000000000000027`. This server is intended to be a testing machine, it has some limitations in comparison to the real machine. This configuration does not provide a PIN login option, it is not possible to update job item and preview of the document is available only in the Y Soft local network. The demo server does not send jobs to print, it only deletes the job from waiting list for a minute and then sends it back to waiting list. Testing of missing functionality is described in section 4.1

Initial data

Two print jobs are sent to the testing server. Both of them are in waiting list. The first one is marked as a favorite. On this server, a testing account is created.

Tested Platforms

- Android 5.1 (Google Nexus 4)
- Android 4.3 (Rockchip tablet)
- Android 4.4 (Samsung M 5370 printer)
- Windows Vista 32bit (Desktop)
- Windows 8.1 (Notebook)
- Windows 10 (Tablet Acer Iconia Tab 8)
- Windows Server 2008R2
- Linux Ubuntu 15.04 (Notebook)
- OS X 10.9.5 Mavericks (Mac Mini)
- OS X 10.11 El Captain (Macbook Pro)

Testing scenario

1. Run the application.
   - Login screen with user name and password input is displayed.
2. Swipe right.
   - Login option is switched to PIN.
3. Swipe Left
   - Login option is switched back to user name and password.
4. Click OK
   - Modal dialog about bad credentials is displayed.
5. Set user name and password according to testing account.
6. Click OK button.
   - Login is successful.
   - Main screen with tabs on the left sites is displayed.

- Print tab is preselected.
- Waiting list is preselected
- First job item is selected as favorites by full start.
- User name is on the top right corner next to the log out button.

7. Click info icon in first job item.
    - Modal dialog is shown
    - Job item information is loaded.
    - Preview is available only in Y Soft local network.
8. Click Change options button.
    - Values of the carousel are preselected according to job options.
    - It is possible to change values by buttons.
9. Click OK button
    - Options are not updated, because of demo server limitations.
10. Select first job using check box.
    - Job is selected
    - Job selection menu is displayed
11. Click Select all button in job selection menu.
    - All jobs are selected.
12. Click Left arrow button in job selection menu.
    - All jobs are unselected.
    - Job selection menu disappeared.
13. Set second job as a favorite using star button.
    - Start is colored.
14. Print first job.
    - Job disappeared from list.
15. Navigate to printed list.
    - Printed list is empty because of demo server limitations.
16. Navigate to favorite list

- List contains two jobs.

17. Select scan tab
    - Tab content is changed by rolling down animation.
    - Tab content contains image describing scan action.
18. Select copy tab
    - Tab content is changed by rolling up animation.
    - Tab content contains image describing scan action.
19. Navigate to billing tab
    - List of two billing codes item is displayed.
20. Click Help tab
    - Tab content contains six help buttons.
21. Click on scanning help button.
    - Internet browser is opened with manual page.

Results

Testing discovered that if it is possible to run the application, then it is possible to go through the use case without a problem. In some cases, it was not possible to run the application. Results of the testing are summarized in table 4.1.

4.1 Other testing

The goal of this section is to test functionality that is not covered by the server as login by PIN, send job for print and update job meta data. This functionality was tested in Y Soft local network on real server. This functionality was tested on Windows 8.1, Linux Ubuntu 15.04, OS X 10.11 El Captain and Android 5.1. All of mentioned features are functional.

Other goal was to find limitations of the lists. In case of more than thirty items in a waiting, printed or favorite list scrolling animation starts to lag. The first hypothesis was that this behavior is caused by images in list items but this disadvantage appears also in simple lists that are contains only a label.

System	Result
Android 5.1 (Google Nexus 4)	Executed without a problem during the use case. One disadvantage are long loading times of the application. Time from application start to the log in screen rendering is about nine seconds.
Android 4.3 (Rockchip tablet)	
Android X.X (Samsung printer)	Same result as on Android 5.1
Windows Vista 32bit (Desktop)	Final package was made on 64-bit environment, it is not possible to run it in 32-bit Windows.
Windows 8.1 (Notebook)	No problems during the use case. The only disadvantage is that application run requires a visible command line window.
Windows 10 (Tablet Acer Iconia Tab 8)	According to Kivy Windows 10 is currently unsupported. It is not possible to run a Kivy application on this system.
Windows Server 2008R2	Running application ends with error: "The application has failed to start because its side-by-side configuration is incorrect."
Linux Ubuntu 15.04 (Notebook)	Linux does not required any portable package. Just run the source code. Testing did not uncover any problems.
OS X 10.9.5 Mavericks (Mac Mini)	Testing did not uncover any problems.
OS X 10.11 El Captain (Macbook Pro)	Testing did not uncover any problems. It is not possible to create the distributable package, but it is possible to run a package created on previous versions-

Table 4.1: Testing Results

4.2 Raspberry Pi

This section is beyond the scope of the thesis. The goal of the following text is to test the application on a Raspberry Pi. For more information on Raspberry Pi, go to the official site [1]. Raspberry Pi is, according to Kivy official documentation, able to run Kivy source code. There are three ways how you can prepare environment and run a Kivy application. First two are by installing Linux (one for Raspbian Jessie and the second for Raspbian Wheezy distribution) and Kivy requirements separately. The third is to download an image containing a preinstalled Kivy Framework and all its requirements.

Within this test, it was decided to use the image with preconfigured system. The specific version of the computer is Raspberry Pi B+. This computer will be controlled by mouse and keyboard connected via the USB port. Internet network is connected by the RJ-45 port.

The goal of the test was to go through the use case described in subsection 4. It was, however, not possible to go through with the testing without first overcoming the following complications. The preinstalled image does not contain the same set of fonts as Kivy on other platforms. This could be fixed by copying font files to the system. Another error that blocked passing the use case was caused by keyboard input. One click on the keyboard wrote two same characters in the text input. This error made logging in impossible. After setting user name and password to text inputs in code, it was possible to go through the use case. List of all discovered errors is summarized in list 4.1

1. Official site of Raspberry Pi: `https://www.raspberrypi.org/`

- Missing default set of fonts on installed image.
- Input one character into the text input widget causes input of two same characters (i.e hit “x” input “xx”)
- Cursor perform clicks on different place than where it is placed (The difference was about twenty pixels on both co-ordinates).
- Values of transition direction were not respected by screen managers. Animation still has the default direction.
- It was not possible to switch to UI with carousel on job detail modal dialog.
- Internet browser was not opened after clicking on one of help buttons.

Figure 4.1: Raspberry Pi Testing Results

Chapter 5

Conclusion

The main goal of this thesis was to find out whether the Kivy Framework is sufficient for commercial development. All of the programs and tools that were used during the development are distributed under the licenses that allows commercial utilization. It is possible to use all procedures described in the thesis in commercial sphere.

One of the main advantages in the research phase of the thesis is that the framework supports multiple platforms. After the testing phase, which included a creation of a distributable package and deployment on the supported operating systems, the value of this advantage degraded. It is necessary to make additional changes in code to make the application run on all officially supported platforms.

The basic set of the widgets provides sufficient functionality to creating a basic GUI. It is possible to extend this functionality to satisfy complex user interface requirements. Architecture of the Kivy Framework is not prepared for a fully localized application. Lack of localization support, described in subsection 2.3.6, is one of the main disadvantages in using the Kivy Framework for larger commercial applications.

It is not possible to access many sensors on mobile devices with the application. Access to devices sensors is described in paragraph 2.1.1. One of the requirement from Y Soft was to analyze the ability to access camera for barcode scanning. Only Android supports taking pictures by a camera.

The next disadvantage is the Kivy support. As it is a community project, the support is provided by community and Kivy developers. Support did not make any commitment on resolving raised issues. This leads to a significant chance for unanswered queries to stall the development process for commercial teams. More on the Kivy support is written in subsection 2.1.2. This means that all of the support of the Kivy Framework in case of a commercial use has to be covered by the developing company.

Kivy is sufficient for creating interactive applications with complex graphic user interface that does not require localization support and access to the device sensors. It is suitable for creating graphic games or utilities for internal purposes. But because of the reasons mentioned before, it is not sufficient for commercial use, especially for creating a terminal application using the Y Soft SafeQ REST API.

Bibliography

[1] Mobile analytics report [online]. https://www.citrix.com/content/dam/citrix/en_us/documents/products-solutions/citrix-mobile-analytics-report-february-2015.pdf, [cit. 2015-09-27].

[2] Building reputation in stackoverflow: An empirical investigation [online]. http://amiangshu.com/papers/msr-challenge-preprint-bosu.pdf, [cit. 2013].

[3] Buildozer [online]. https://github.com/kivy/buildozer, [cit. 2015].

[4] Groups [online]. http://learn.googleapps.com/products/groups, [cit. 2015].

[5] Gstreamer application development manual (1.6.0) [online]. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/html/index.html, [cit. 2015].

[6] Kivy designer [online]. https://github.com/kivy/buildozer, [cit. 2015].

[7] Kivy documentation [online]. http://kivy.org/docs/, [cit. 2015-04-11].

[8] Kivy homepage [online]. http://kivy.org/, [cit. 2015-04-11].

[9] Kivy on raspberry pi [online]. http://kivypie.mitako.eu, [cit. 2014].

[10] Kivy public source code [online]. https://github.com/kivy, [cit. 2015-07-07].

[11] A.M.S. Laurent. *Understanding Open Source and Free Software Licensing.* O'Reilly Media, Inc., 2004.

[12] Offline editor wiki pro android [online]. http://www.fit.vutbr.cz/study/DP/BP.php?id=14292, [cit. 2013].

[13] Plyer [online]. https://github.com/kivy/plyer, [cit. 2015].

[14] Pv248 python [online]. https://is.muni.cz/predmet/fi/podzim2014/PV248?lang=en, [cit. 2014].

[15] Pyinstaller [online]. http://www.pyinstaller.org/, [cit. 2015].

[16] Pyjnius [online]. https://github.com/kivy/pyjnius, [cit. 2015].

[17] Pyobjus [online]. https://github.com/kivy/pyobjus, [cit. 2015].

[18] Python [online]. https://www.python.org, [cit. 2015].

[19] Python for ios [online]. https://github.com/kivy/kivy-ios, [cit. 2015].

[20] Should i use python 2 or python 3 for my development activity? [online]. https://pypi.python.org/pypi/3to2, [cit. 2013-04-16].

[21] Tools and techniques for creating mobile applications [online]. http://www.fit.vutbr.cz/study/DP/BP.php.en?id=15155, [cit. 2013].

[22] Robert W. Sebesta. *Concepts of Programming Languages.* Addison Wesley Longman, 4 edition, 1999.

[23] Xcode project [online]. https://developer.apple.com/library/ios/featuredarticles/XcodeConcepts/Concept-Projects.html, [cit. 2011].

Appendix A

Attachment Content

As a part of the thesis, it contains:

- Source code in LaTeX format and final version of thesis as a portable document format (pdf).
- Source code for developed application.
- The latest version of portable packages of the Kivy Framework.
- Screen shots including all provided versions of mockups.
- Text file containing abstract text.

Appendix B

Appendix

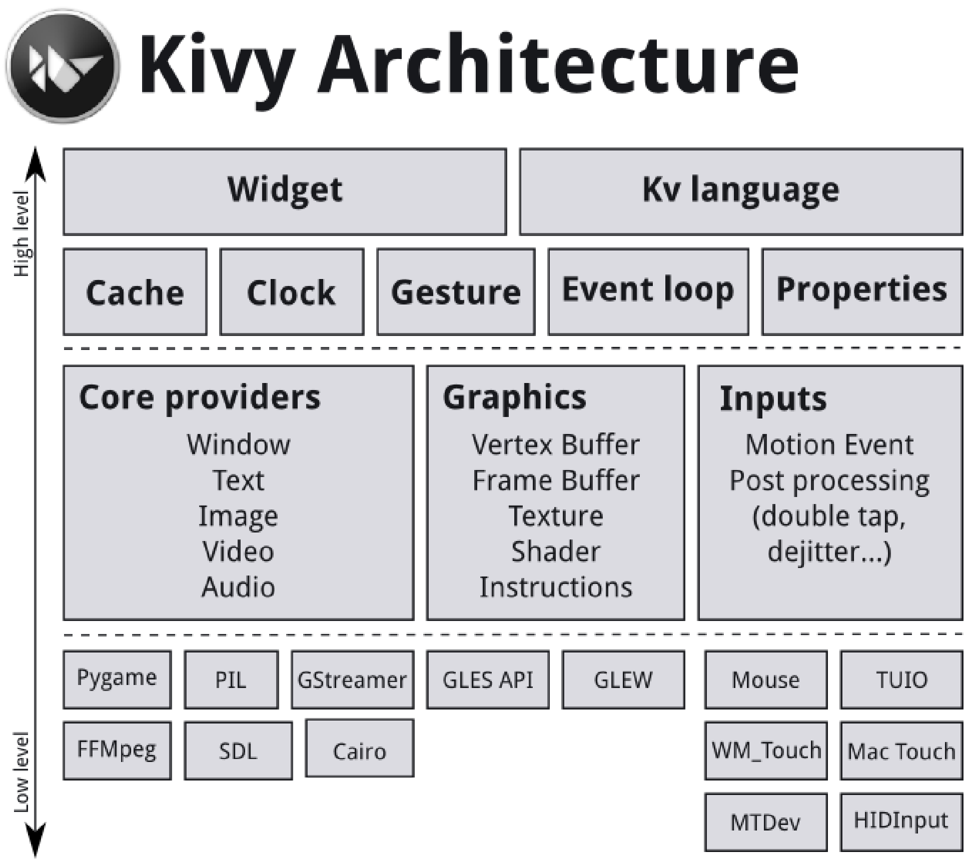


Figure B.1: Kivy three layer architecture devided into build blocks.

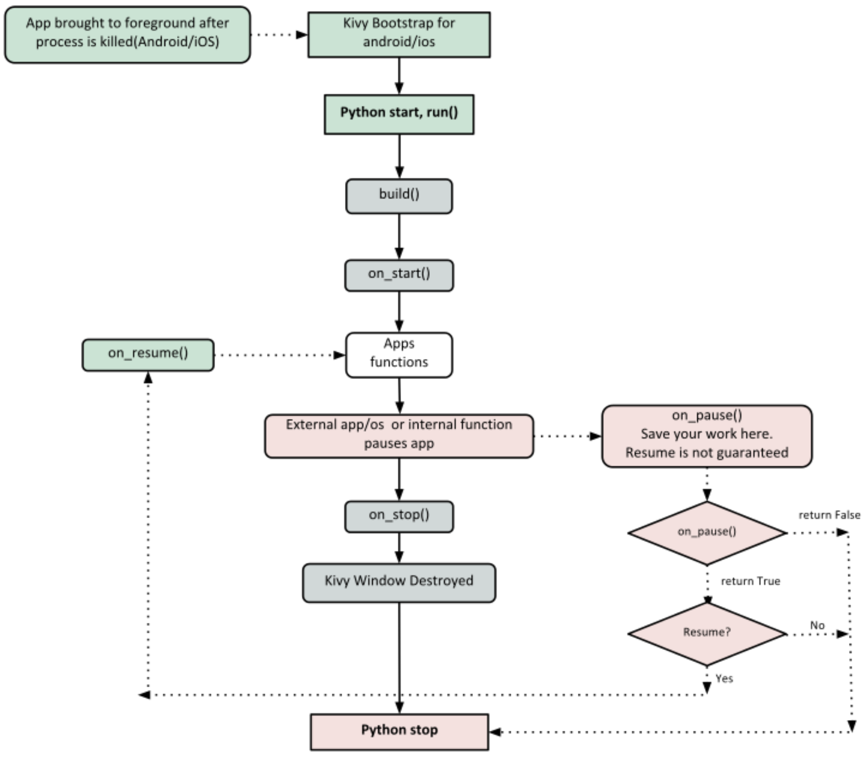

App brought to foreground after process is killed(Android/iOS)
Kivy Bootstrap for android/ios
Python start, run()
build()
on_start()
on_resume()
Apps functions
External app/os or internal function pauses app
on_pause() Save your work here. Resume is not guaranteed
on_stop()
on_pause()
return False
Kivy Window Destroyed
return True
Resume?
No
Yes
Python stop

Figure B.2: Livecycle of a Kivy application with API for saving and loading state [7].

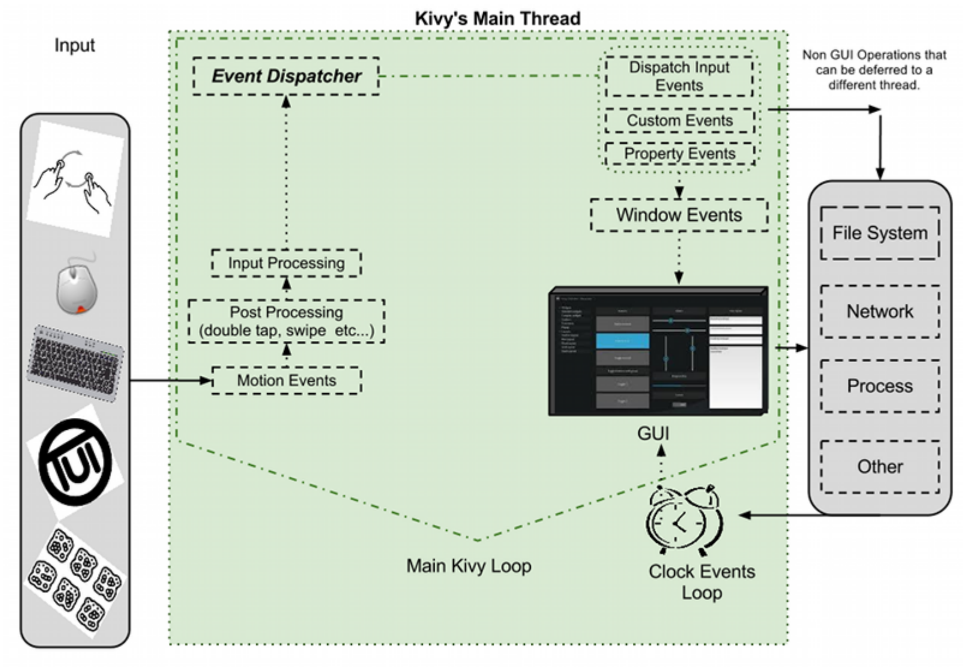


Figure B.3: Diagram describing processing user interaction events in a loop [7].

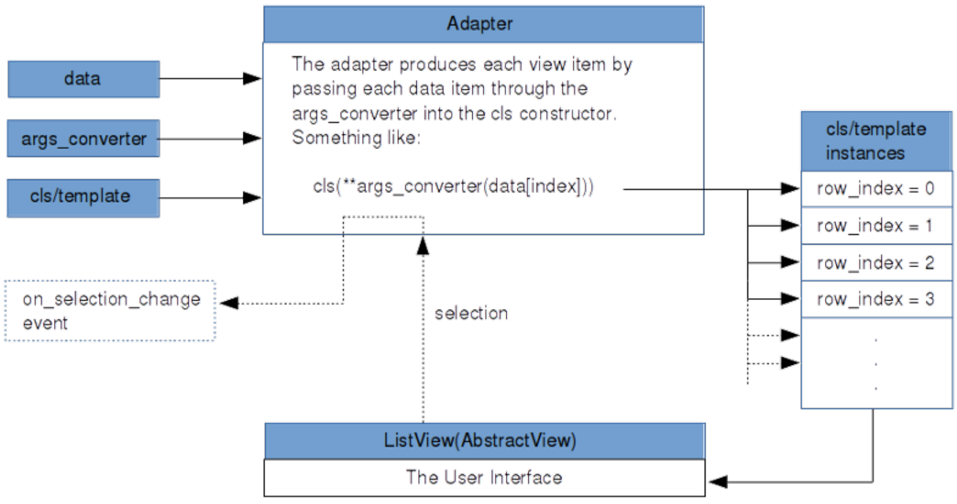

Adapter
The adapter produces each view item by passing each data item through the args_converter into the cls constructor. Something like:
cls(**args_converter(data[index]))
data
args_converter
cls/template
on_selection_change event
selection
cls/template instances
row_index = 0
row_index = 1
row_index = 2
row_index = 3
ListView(AbstractView)
The User Interface

Figure B.4: List Adapter Principle

Platform	Android	iOS	Windows	OS X	Linux
Accelerometer	X	X		X	X
Call	X				
Camera (taking picture)	X				
GPS	X	X			
Notifications	X		X	X	X
Text to speech	X	X	X	X	X
Email (open mail client)	X	X	X	X	X
Vibrator	X	X			
Sms (send messages)	X				
Compass	X	X			
Unique ID	X	X	X	X	X
Gyroscope	X	X			
Battery	X	X	X	X	X
Native file chooser			X	X	X
Orientation	X				
Audio recording	X				
Flash	X	X			

Table B.1: Plyer Supported API [13]