**F21SC:**

**Industrial**

**Programming**

*Coursework 2*

*Students:*

Eliott BLONDIN

[*eb82@hw.ac.uk*](mailto:eb82@hw.ac.uk)

Quentin DUCASSE [*qd14@hw.ac.uk*](mailto:qd14@hw.ac.uk)

*Lecturer:*

Hans-Wolfgang LOIDL



Table of Contents

[Introduction 4](#_Toc26396721)

[User Guide 4](#_Toc26396722)

[Developer Guide 5](#_Toc26396723)

[Testing 12](#_Toc26396724)

[Reflections on Programming Language and Implementation 13](#_Toc26396725)

[Conclusions 15](#_Toc26396726)

[References 16](#_Toc26396727)

# Introduction

The given report has for objective to provide a deeper understanding of the coursework in several ways. First, a *Requirement’s Checklist* is provided, showing the specifications along with their fulfilment state. The specifications are extracted from the subject of the coursework and, for some, modified or added. They provide an overview of the state of the application. Next come the *Design Considerations*, reflections on the choice of the design of both the backend project, the frontend project as well as the solution structure. The two next parts are practical and aims at the user through the *User Guide* and the developer that would like to understand or enhance the actual application in the *Developer Guide.* The next part covers *Testing* and mainly presents the Unit Tests and their coverage. In the end are provided *Reflections on Programming Language and Implementation* which purpose is to rate the choice of the language features and technologies that are fitting the needs of the application as well as comparing it to other type of languages.

The coursework has for objective to make us create from scratch and from a professional IDE a simple application with a GUI. Coming from a Java/Python background and having Object-Oriented Programming knowledge before this task, I was looking into the coursework with two main fears:

* How to adapt my previous knowledge to a new language?
* How to use the Visual Studio IDE along with a GUI designer on a computer it was not designed for (macOS 10.14)?

The first one was simple to overcome as C# is very similar to Java and Visual Studio provides an intuitive syntax correction the same way Eclipse would with Java. The properties and language features of C# were explained through the lectures and were easy to put the hands on. However, the second point resulted in some GUI-related issues as well as complicating the obtention of a standalone executable. Those issues will be explained further into the report.

The coursework allowed me to challenge myself in reusing the knowledge I had as well as to adapt to a new syntax, language features and IDE. Overall, I found it interesting and rewarding in the end.

# User Guide

***Installation and Usage***

The project can be installed with the source code using the directory hosted on Github at the following URL, <https://github.com/QDucasse/IssuuTracker>. Instructions about installation are hosted as well but here are the main steps:

*Installing the project:*

bash command

$ python setup.py install

*Running the Command Line Interface:*

The help contained in the Command Line Interface can be shown using the following command.

bash command

$ python cli.py --help

usage: cli.py [-h] [-u USER\_UUID] [-d DOC\_UUID] [-t TASK\_ID] [-f FILE\_NAME]

ISSUU Tracker

optional arguments:

-h, --help show this help message and exit

-u USER\_UUID, --user\_uuid USER\_UUID

UUID of the user

-d DOC\_UUID, --doc\_uuid DOC\_UUID

UUID of the document

-t TASK\_ID, --task\_id TASK\_ID

ID of the task (2a,2b,3a,3b,4d,5,6)

-f FILE\_NAME, --file\_name FILE\_NAME

JSON file of the dataset

The main option is the task id that represent one of the requirements stated in the subject of the coursework. The tasks correspond to the following:

* Task 2a: Plots the user’s home countries, requires a file (-f).
* Task 2b: Plots the user’s home continents, requires a file (-f).
* Task 3a: Plots the user’s user agent (full), requires a file (-f).
* Task 3b: Plots the user’s user agent (trimmed), requires a file (-f).
* Task 4: Plots a list of documents UUID that can be “also liked”, requires a file (-f), a document UUID (-d) and optionally a visitor UUID (-u).
* Task 5: Prints the graph of the “also liked” documents, requires a file (-f), a document UUID (-d) and optionally a visitor UUID (-u).
* Task 6: Launches the GUI, requires no other arguments.

For example, running the task 5 with the flag -t 5 on the dataset issuu\_cw2.json with the user UUID value 2f63e0cca690da91 and the document UUID value 140219141540-c900b41f845c67cc08b58911155c681c can be done as follows:

bash command

$ python cli.py -t 5 \

> -f issuu\_cw2.json \

> -u 2f63e0cca690da91 \

> -d 140219141540-c900b41f845c67cc08b58911155c681

# Developer Guide

The project code is clearly split between the different classes as explained in the *Design Considerations* part. This is a clear advantage as it allows a clear insight of the structure of the project through its imports. The important methods in each of the classes are presented and discussed but the whole code fully commented is available at <https://github.com/QDucasse/IssuuTracker>. Moreover, the environment in which the project has been developed will be discussed.

***Development Environment***

The whole project was developed with the help of a *virtual environment*. The idea behind a virtual environment is to control what is currently loaded using pip (packages and libraries) in order to control portability and to allow the application to be distributed easily. In order to use a *virtual environment*, we chose to use virtualenvwrapper [REF], a wrapper around venv that helps creating and managing *virtual environments*. In order to create a *virtual environment*, we first need two files: requirements.txt holding the needed packages and libraries for the application as well as setup.py allowing a simple installation of the application. The files are showed as follows:

requirements.txt

matplotlib

graphviz

pytest

setup.py

setup(

name='IssuuTracker',

description='Coursework2-Industrial Programming',

long\_description=long\_description,

  url='https://github.com/QDucasse/IssuuTracker',

# Author details

author='Quentin Ducasse',

author\_email='qd14@hw.ac.uk, eb82@hw.ac.uk',

license='MIT',

  classifiers=[

'Development Status :: 3 - Alpha',

'Intended Audience :: Education',

'Topic :: Scientific/Engineering',

'Topic :: Scientific/Engineering :: Programming',

'Topic :: Scientific/Engineering :: Data Science',

'Topic :: Utilities',

'License :: OSI Approved :: MIT License',

'Programming Language :: Python :: 3.7',

'Operating System :: Apple :: OSX'

],

packages=find\_packages(),

install\_requires=['matplotlib','graphviz','pytest'],

entry\_points={"console\_scripts": ["qducasse=IssuuTracker.\_\_main\_\_:main"]},

)

Those two files allow the creation of a *virtual environment* with the following commands. The first one installs the package through pip while the second creates the environment itself while in the folder of the project.

bash command

$ pip install virtualenvwrapper

bash command

$ mkvirtualenv -a . -r requirements.txt <venv name>

We can now access the *virtual environment* by using the following command. The name of the *virtual environment* should be shown between parenthesis to show the one we are currently in. Using the command deactivate will stop the current *virtual environment*.

bash command

$ workon <venv name>

(<venv name>) $ ...

Installing the project the way it would be installed by other users can be done using pip install -e . which installs the current package in editable mode. Finally, the standalone executable was created with the pyinstaller [REF] plugin using the following command while outside any virtual environment:

bash command

$ pyinstaller cli.py -n cw2 \

--add-binary='/System/Library/Frameworks/Tk.framework/Tk':'tk' \

--add-binary='/System/Library/Frameworks/Tcl.framework/Tcl':'tcl' \

--hidden-import='tkinter' --onefile

***Class examples***

If the global aspect of the different components has been discussed in the *Design Consideration* part, we will now dive into the important methods of each class. Each class (or .py file) has some basic behaviour that can be displayed by executing the file itself. This is done by using the line:

\*.py file

if \_\_name\_\_==”\_\_main\_\_”:

...

The code after this line is only executed if the current file is being executed, and therefore will not be executed if the file is being imported into another for example. This feature is extremely handy to come with a quick prototype or provide an example of the class being implemented in the current file. Moreover, it is more readable than unit tests and provide a quick look into the API of the class being defined.

***Project Walkthrough***

The project is split between the following classes and we will present the main methods of each of those classes in this part:

* ***Data Loader***
* ***Continent Converter***
* ***Data Visualiser***
* ***Affinity Finder***
* ***Graph Handler***
* ***GUI***

We will pass through the most important methods of each of the classes of the application and point out interesting behaviour or design. The docstrings will be removed due to space constraints but can be seen in the source code or on Github.

***ContinentConverter***

The ContinentConverter class is a helper providing a dictionary holding countries 2-letters code and their corresponding continent. We planned on using pycountry-convert as an external library to perform this job, but we did not need all of the implemented functions. Therefore, only the dictionary and one function have been taken from the package and included in our application as a component.

continent\_converter.py

class ContinentConverter():

COUNTRY\_ALPHA2\_TO\_CONTINENT = {

'AB': 'Asia',

'AD': 'Europe',

'AE': 'Asia',

'AF': 'Asia',

'AG': 'North America',

'AI': 'North America',

'AL': 'Europe',

...

'YT': 'Africa',

'ZA': 'Africa',

'ZM': 'Africa',

'ZW': 'Africa',

}

def convert\_country\_alpha2\_to\_continent(self,country\_2\_code):

if country\_2\_code not in self.COUNTRY\_ALPHA2\_TO\_CONTINENT:

return 'Undefined'

return self.COUNTRY\_ALPHA2\_TO\_CONTINENT[country\_2\_code]

This one function uses the dictionary to return the continent the country which country code is provided is from. In cases where the country code is not defined, it returns “Undefined”. This case was added to the code taken from pycountry-convert as it is needed in our dataset (some codes are “ZZ”, “EU” or “AP” and are not defined officially).

***DataLoader***

The DataLoader is the interface the application has with the dataset in terms of loading and performing operations on its attributes. The main functions of the loader are loading the dataset, adding a “*continent*” field to the instances and adding a “*trimmed user agent*” field to the instances. Loading the file is done by evaluating it line by line and adding the dictionary to the instance variable dicts of the DataLoader.

data\_converter.py

def load\_dataset\_from(self,path):

with open(path,'r') as file:

for line in file:

self.dicts.append(eval(line))

Then, for the operations on the fields of the dictionary, the idea is the same for the “*continent*” and “*trimmed user agent*”. An action is performed on one of the fields (either getting the continent from the visitor\_country field or trimming the visitor\_user\_agent field) and it is mapped among all the dictionaries present in the dicts instance variable. We will look at the example of the “browser trimming” as the continent example simply uses the ContinentConverter method.

data\_converter.py

def add\_trimmed\_browser(self,dict):

pattern = '([a-zA-Z]\*)\/'

if 'visitor\_useragent' in dict:

browser\_verbose = dict['visitor\_useragent']

if re.match(pattern,browser\_verbose):

dict['visitor\_useragent\_trimmed'] = re.findall(pattern,browser\_verbose)[0]+' '+re.findall(pattern,browser\_verbose)[-1]

else:

dict['visitor\_useragent\_trimmed'] = dict['visitor\_useragent']

return dict

A regular expression is used to get the new name of the user agent. It matches the words followed by slashes and considers the new user agent to be the concatenation of the first one and last one. For example, “Mozilla/5.0 (Windows NT 6.0) Chrome/33.0.1750.117 Safari/537.36” will output “Mozilla Safari”. Before doing this matching, the key is checked to be in the dictionary and if the regex does not match, the full visitor user agent is used. We made this choice because in absence of matching, the user agent is usually short.

data\_converter.py

def map\_trim(self,dicts=None):

if dicts is None:

dicts = self.dicts

dicts\_with\_trim = [self.add\_trimmed\_browser(dict) for dict in self.dicts]

self.dicts = dicts\_with\_trim

List comprehension is used to map the above function on all the dictionaries within the given list of dictionaries (the instance variable by default). The same functions are written for the “*continent”* field using the cconv instance variable, an instance of ContinentConverter.

***DataVisualiser***

The DataVisualiser is composed of one core method and several usages of it. The core method plots one field of a given list of dictionaries. The plot is a histogram counting the different appearances of a value for a given field.

data\_visualiser.py

def plot\_feature(self,dicts,feature,xlabel,ylabel):

feature\_dict = {}

for dict in dicts:

if dict[feature] in feature\_dict:

feature\_dict[dict[feature]] += 1

else:

feature\_dict[dict[feature]] = 1

plt.figure()

plt.bar(list(feature\_dict.keys()),list(feature\_dict.values()))

plt.xlabel(xlabel)

plt.ylabel(ylabel)

plt.tick\_params(labelsize=5)

plt.show()

The first part of the method creates a dictionary holding values of the given feature as keys and the number of representations of that value as values. This dictionary is then plotted using the lists of keys and values put into a plt.bar. This method is then reused in order to plot the different features.

data\_visualiser.py

def plot\_countries(self,dicts=None):

if dicts is None:

dicts = self.dicts

self.plot\_feature(dicts,'visitor\_country','Country','Count')

***AffinityFinder***

The AffinityFinder is the component of the application that has to compute the different metrics on documents and users UUIDs. The three core functions it implements are has\_read(user\_uuid), readers\_of(doc\_uuid) and also\_likes(doc\_uuid). These methods provide metrics on other users or documents of a given user or document.

data\_visualiser.py

def readers\_of(self,doc\_uuid,dicts=None):

if dicts is None:

dicts = self.dicts

return list(set([dict['visitor\_uuid'] for dict in dicts if(('subject\_doc\_id' in dict) and ('visitor\_uuid' in dict) and (dict['subject\_doc\_id']==doc\_uuid))]))

The last line and most important one (even though it seems unreadable here) uses list comprehension in order to get the readers of a given document UUID. The first two conditions check for the existence of the different keys in the given dictionary. It translates to:

Translation of data\_visualiser.py

readers\_of = []

for dict in dicts:

if(('subject\_doc\_id' in dict) and ('visitor\_uuid' in dict)):

if (dict['subject\_doc\_id']==doc\_uuid):

readers\_of.append(dict['visitor\_uuid'])

return list(set(readers\_of))

The reverse process is done for the has\_read() method:

data\_visualiser.py

def has\_read(self,visitor\_uuid,dicts=None):

if dicts is None:

dicts = self.dicts

return list(set([dict['subject\_doc\_id'] for dict in dicts if(('visitor\_uuid' in dict) and ('subject\_doc\_id' in dict) and (dict['visitor\_uuid']==visitor\_uuid))]))

Finally, the also\_likes() method combines the two and adds two important aspects. First of all, the visitor UUID is optional, but when given, the function will not take in consideration the documents already read by this user. In concrete words, this means a given user will not be interested in books he has already read, and they do not need to be included in its recommendations. Another important point is that a sorting function can be provided to the also\_likes() method. This function will output the list in a special order. The default one will return the 9 most read other documents and add the input document UUID to it. This choice was made over simply returning the 10 most read because in graph creation, the input document could sometimes not be displayed as it has not been read enough times.

data\_visualiser.py

def also\_likes(self,doc\_uuid,visitor\_uuid=None,dicts=None,sort\_func=None):

if dicts is None:

dicts = self.dicts

if sort\_func is None:

sort\_func = self.sort\_10best

if visitor\_uuid == '':

visitor\_uuid = None

dicts\_list = [docs for visitor in self.readers\_of(doc\_uuid,dicts) for docs in self.has\_read(visitor,dicts) if visitor!=visitor\_uuid]

return sort\_func(dicts\_list,doc\_uuid)

As in the two above methods, this one uses list comprehension to translate the behaviour detailed underneath:

Translation of data\_visualiser.py

also\_likes = []

for visitor in self.readers\_of(doc\_uuid,dicts):

if visitor != visitor\_uuid:

for doc in self.has\_read(visitor,dicts):

also\_likes.append(doc)

return sort\_func(also\_likes)

Finally, the sorting function used as the default one is the following. It sorts the elements by the number of appearances they have then remove duplicates as well as the given document UUID and finally outputs the given document UUID concatenated to the 9 most read documents.

data\_visualiser.py

def sort\_10best(self,docs\_list,doc\_uuid):

counts = Counter(docs\_list)

sort\_list = sorted(docs\_list, key=counts.get, reverse=True)

sort\_list = list(set(sort\_list))

if doc\_uuid in sort\_list:

sort\_list.remove(doc\_uuid)

return [doc\_uuid] + sort\_list[:10]

***GraphHandler***

The GraphHandler is the wrapper around the graphviz package in order to generate automatically graphviz graphs based on the also\_likes() functionality. The node creation is separated between users and documents to differentiate the shape and add edges between users and documents when adding a new document node.

data\_visualiser.py

def create\_visitor\_node(self,visitor\_uuid,graph=None,base\_visitor\_uuid=None):

graph.attr('node',shape='square')

if visitor\_uuid == base\_visitor\_uuid:

graph.attr('node',style='filled',color='green')

graph.node(visitor\_uuid[-4:])

graph.attr('node',style='solid',color='black')

data\_visualiser.py

def create\_document\_node(self,document\_uuid,visitor\_uuid,graph=None,base\_document\_uuid=None):

if base\_document\_uuid is None:

base\_document\_uuid = self.base\_document\_uuid

graph.attr('node',shape='circle')

if document\_uuid == base\_document\_uuid:

graph.attr('node',style='filled',color='green')

if not(document\_uuid in graph.node\_attr):

graph.node(document\_uuid[-4:])

graph.edge(visitor\_uuid[-4:],document\_uuid[-4:])

else:

graph.edge(visitor\_uuid[-4:],document\_uuid[-4:])

graph.attr('node',style='solid',color='black')

Both methods are changing the shape of the node it will create and change the colour if it is the given document or user UUID. The document adds edges and checks for their existence beforehand.

data\_visualiser.py

def create\_graph(self,dicts=None,base\_visitor\_uuid=None,base\_document\_uuid=None):

if dicts is None:

dicts = self.dicts

if base\_visitor\_uuid is None:

base\_visitor\_uuid = self.base\_visitor\_uuid

if base\_document\_uuid is None:

base\_document\_uuid = self.base\_document\_uuid

self.graph.name = 'al'+base\_document\_uuid[-4:]

al = self.af.also\_likes(doc\_uuid=base\_document\_uuid, visitor\_uuid=base\_visitor\_uuid)

visitors = self.af.readers\_of(doc\_uuid=base\_document\_uuid,dicts=dicts)

self.graph.attr('node', shape='square')

for visitor in visitors:

self.create\_visitor\_node(visitor,self.graph,base\_visitor\_uuid)

  docs = self.af.has\_read(visitor\_uuid=visitor,dicts=dicts)

for document in docs:

if document in al:

self.create\_document\_node(document,visitor,self.graph, base\_document\_uuid)

self.graph.render('./graphs/'+self.graph.name,view=True)

return self.graph

This function uses the also\_likes() function created before then iterates through the readers\_of() and has\_read() starting from the given document UUID. The list created by also\_likes() is used to display only the documents found within it. This allows to limit the number of documents in the graph to the 9 most read and the input document but results in sometimes having a visitor that has only read the input document (see image below).



The visitors in red are not useful to the graph but will appear anyway. They read documents that are not displayed because less read than the 10 actually displayed.

***GUI***

The GUI is composed of four different frames:

* *Data loading:* Choosing the dataset to load
* *Statistics:* Displaying the plots on countries, continents, etc.
* *UUIDs:* Entries to input the UUIDs
* *Also Likes:* List of the UUIDs “also liked” or graph of them

***Main***

The Main part implements an ArgumentParser to display a help to the user and handle the different arguments that can be supplied to it. This part is a wrapper around the other components.

# Testing

Testing is done on every component of the application using pytest [REF]. Testing is often limited as most of the results are displayed, especially for the DataVisualiser, GraphHandler or GUI. However, it is possible to test the histogram in forms of dictionary for the DataVisualiser and test the content of .dot files for the GraphHandler. The whole tests suite can be run with:

bash command

$ pytest

======================== test session starts =========================

collected 19 items

IssuuTracker/tests/test\_affinity\_finder.py ..... [ 26%]

IssuuTracker/tests/test\_continent\_converter.py .. [ 36%]

IssuuTracker/tests/test\_data\_loader.py ....... [ 73%]

IssuuTracker/tests/test\_data\_visualiser.py .... [ 94%]

IssuuTracker/tests/test\_graph\_handler.py . [100%]

========================= 19 passed in 0.59s =========================

Special datasets have been created to perform tests on and can be found in the data/tests folder. These datasets only contain the useful parameters for each dictionary and the current component being tested. For example, the isssuu\_test\_data\_loader.json only contains dictionaries with the visitor\_country and visitor\_useragent fields.

issuu\_test\_data\_loader.json

{"visitor\_useragent":"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/33.0.1750.117 Safari/537.36","visitor\_country":"MX"}

{"visitor\_useragent":"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/540.36 (KHTML, like Gecko) Chrome/34.0.1750.112 Safari/538.36","visitor\_country":"MX"}

{"visitor\_useragent":"P3P Validator","visitor\_country":"FR"}

For the AffinityFinder and GraphHandler, a dataset is composed of the dictionaries represented by the following graph and with the fields visitor\_uuid and subject\_doc\_id only.

issuu\_test\_affinity\_finder.json

{"visitor\_uuid":"0000","subject\_doc\_id":"aaaa"}

{"visitor\_uuid":"1111","subject\_doc\_id":"aaaa"}

{"visitor\_uuid":"1111","subject\_doc\_id":"bbbb"}



# Reflections on Programming Language and Implementation

Based on our experience in implementing this application, we think that the features and technologies that have been the most useful to the development are the following.

First, considering language features, *List Comprehension* is an extremely handy feature as it manages the role of several loops within a line. It allows to think quicker about the content of the final list rather than working around indentations and conditions.

Now, considering the kind of language, Object-Oriented Programming was particularly interesting here as it helped split the application in several components and projects. This allows to test them separately to assert their standalone behaviour as well as integrate them accordingly. This was particularly helpful to develop the backend/frontend relations and to test the backend part accordingly. The use of Object-Oriented Programming allows to reuse Design Patterns as well that are proven to work. This is where I tried to look upon the MVC (Model View Control) or MVVM (Model View View Model). This allows the use of effective designs that are proven to be effective at what they do.

Now if we look at the limitations of my application, the flaws are mostly in the GUI as it feels pretty rigid and dated even though it is functional. One of those flaws is the fact that once the user has chosen the bookmark or previously visited site he wants to go to; he has to press go and the search bar now contains the URL. The URL is in the search bar, sure, but the user still has to press search in order for the HTTP request to be launched. This is mostly because embedding the search within the other windows would result in a loss of focus from the GUI and the main window crashing. The GUI feels less dynamic but is more reliable.

The use of a scripting language for the design of the application certainly is possible with Perl for example using Tk or even GTK as the GUI designer. The result should be lighter than the current project as this one is heavily structured. However, the relations between the objects are minimal and very little proper Object-Oriented Programming features are used (Inheritance, Interfaces, Polymorphism, etc.). The final solution is still condensed using C# and allows reusability. Application built with C# should suffer from the OOP issues of being heavy in lines of code but allow reusability and clear code separation on the other hand.

# Conclusions

In the end, this development experience was interesting because it was the first contact we had to Python that is widely used in the industry. There are some particular points that we are proud of in the end as well as some we would have liked to have done differently.

***Lessons from CW1***

The feedback from the Coursework 1 certifies the feeling we had that there is a need for the project structure to come along with the design considerations as there is a clear need to split properly the different packages and classes to test them better and certify them individually. This aspect brings reliability inside the project. We tried to push the *Developer’s Guide* more by diving more in depth into methods descriptions to provide a good understanding of the project without looking at the source code.

***Pros***

First of all, we find the structure of the project satisfying both in the class design and project structure by having everything separated. I think it allows for reusability and more accurate testing. We think the unit tests bring more reliability to the different components and allow to debug more precisely. We tried to use a Test-Driven Development approach even though we dropped this point of view while arriving in the different part requiring a control over what is displayed (DataVisualiser, GraphHandler, GUI). Finally, all the code is commented with proper docstrings and example are provided within the if \_\_name\_\_ == “\_\_main\_\_”: part of each file. The use of virtual environments and a proper project structure gives our project a way to install it easily and to distribute it without looking at the OS or previously installed packages and libraries.

***Cons***

The GUI we finally created is rather static and does not feel really handy but remains practical and provides all the functionalities it is supposed to have. The parsing of the different arguments provided to the *Command Line Interface* is not complete as well and would have needed more checks on the arguments passed. Finally, the tests are definitely not covering enough of the application. We could not reproduce the different graphs provided as more readers and documents were found and displayed.

***Deliverables***

The deliverables for this coursework consist of this report, the source code and a standalone executable. The standalone executable and source code are given along with the report. The source code is also published on GitHub under the <https://github.com/QDucasse/IssuuTracker> URL. A live demo of the application can be seen on <https://youtu.be/otgHuG6koHY>.

***Contribution***

We believe to have contributed nearly equally on the project and report.

# References

[1]: *Introduction to Model View Control (MVC) Pattern using C#***[visited on October 31st]** <https://www.c-sharpcorner.com/article/introduction-to-model-view-control-mvc-pattern-using-C-Sharp/>

[2]: *Model-View-Viewmodel* **[visited on October 31st]** <https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm>

[3]: *Stack overflow forum “How to get website title from c#”* **[visited on October 31st]** <https://stackoverflow.com/questions/329307/how-to-get-website-title-from-c-sharp>

[4]: *The Perfect URL Regular Expression* **[visited on October 31st]** <http://urlregex.com/>

[5]: *John K. Ousterhout: “Scripting: Higher-Level Programming for the 21st Century” –*

*March 1998*