1 Introduction

This paper will explore the architecture of the flask[1] web framework. Flask is a lightweight web application framework, based on WSGI[2]. as explained on their github repository. The code analysis will be done by constructing two small scripts that extract relevant data from the target framework. These two scripts will be built through this paper in an evolutionary development process, where the paper will highlight the intermediate steps.

The reason for this evolutionary practice is two fold, both to enlighten the reader on the development process and how python can be utilizing for this intent, but also to show the architectural process with and the aspects revealed about the flask framework. The code analysis will be done in python utilizing reflection[3], and combine both dynamic and static code analysis, in this way. This paper is, however, just one way of utilizing reflection in python, and leaves it out of scope to evaluate on the type of reflection utilized, but will briefly mention other types as a future topic of study. Finally it will conclude on the findings of the flask framework, the value of the autogenerated graphs, as well as utilizing reflection to do architectural reconstruction.

2 Problem Elicitation

An architectural reconstruction process is initialized by a problem from faced by stakeholders. This is naturally difficult, as I operate as an external actor without prior knowledge of the system or connection to any known stakeholders. For these reason, the stakeholder and problem analysis part has been left out, and I instead create an imagined problem.

Multiple opensource frameworks have a great documentation for usage, but poor documentation for maintainers. Architecture and similar aspects of the development process is often done by accident or at the very least not done explicitly. This makes it difficult for new maintainers to join large projects, as they often have large code bases, which can be difficult to get a fulfilling overview of.

However, maintaining a great architectural model of a system can be expensive and take time, especially in an often underfunded opensource project. Therefore automation is crucial for such a model to continuously be up-to-

[1]`https://flask.palletsprojects.com/en/1.1.x/`
[2]`https://wsgi.readthedocs.io/en/latest/index.html`
[3]`https://wsgi.readthedocs.io/en/latest/index.html`

date with the state of the codebase.

Based on these considerations the following two problems has been decided upon.

> As a **potential future maintainer**, I would like to get an overview of the system, so I understand the system to a degree that makes it possible to solve issues.
>
> As an **active maintainer**, I want any architectural document to be fully automated, so I don't need to update it when changing the source code, which I might forget or discourage me from developing the system.

2.1 Project choice

After quickly brainstorming for various interesting frameworks, preferably build around python as that is, to date, my favorite programming language, I decided upon using Flask as my target framework. Flask is, in my opinion, an interesting choice for a few different reasons:

Lacks documentation As is often the case in projects of this type,Flask has a great documentation for how to usage but are less documented from a contributors point of view. Flask does have a great level of inline comments, making it easier to understand each component of the system, and these comments are transpiled into an online documentation, however understanding the context of each element is difficult, and requires a throughout reading of the source code at the current point in time.

Broad adoption It is used by a lot of different developers across the world. It has fifty thousand stars on github[4], which generally speaking only reflects a small subset of active users, as I, for instance, have used flask multiple times but haven't given it a star on github. Having a large user group makes the project more interesting and makes the results both interesting and relevant. Also for the simple reason that it is interesting to learn the inner workings on a broadly adopted framework.

Size Flask is, compared to other python web frameworks, relatively small. This makesit easier to draw readable diagrams, without being cumbered by the inherit size of the framework. The framework is, however,

[4]`https://github.com/pallets/flask`

still of a size that it is difficult to fully grasp simply by looking at the source code, making it ideal for study, as it gives reasonable challenges.

Person interest I've used flask a multitude of times and would love to learn how it works. It is also a great example as a sort of proof-of-concept of recovering the architecture of an open-source project, which might be an interesting study to delve into at a later point.

Based on these considerations I chose the Flask framework.

3 Selecting the views

As previously mentioned this process will be artificial in nature, and primarily focused on my assumptions of the needs of a maintainer. The choices are based on on my prior experiences with exploring and familiarizing myself with any new framework or technology. The choices are based on the 3+1 architectural model[5], but also inspired by the 4+1 model[6], as these are similar in nature. The naming will be used I have chosen to produce the following views, based on such considerations:

Component Diagram This view will map each module of the framework, and how these are interdependent. It shall include direct external dependencies, so any module that any part of the `Flask` framework explicitly includes, however any second-degree dependency will be ignored, to keep the view simpler in nature. This will provide an abstract overview of the system, and should detail how various parts of the system work together, to understand the areas of responsibility as well as the overall structure and dependencies of these modules.

Class Diagram The development view is essentially abstract in nature, and lack the concrete classes and makes it difficult to actually know where various classes lie. A logical view provides a closer look at the classes, and how these relate to eachother, be in inheritance or association. The current API document of flask[7], contains a list of the most important classes, but it is difficult to fully grasp how these fit together, nor get a quick overview of the relations without jumping back and forth in the documentation.

[5]An Approach to Software Architecture Description Using UML (Revision2.0)
[6]`http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf`
[7]`https://flask.palletsprojects.com/en/1.1.x/#api-reference`

These two diagrams cover the *Development View* and the *Logical View* of the 4+1[6] model, and are both covered in the *Module View* of the 3+1 architectural model[5].

This report will elude any target views, as the goal of the reconstruction will be the source views, which makes this report diverge from the Symphony Model[8], that is normally used in architectural recovery. The reasoning largely stems from the fact that the model does not fully fit the current usecase fully.

3.1 The tools and work process

The following parts of the paper, will show various code snippets and graphs.

All code snippets will be written in Python3.8, however should work on any Python version greater than Python3.6. The paper will first illustrate the snippet and below it will show the output of said snippet. When the python code is used to generate graphs, it will elude the textual output of the snippet, and simply show the visual representation of the graph.

The graphs will be generatted using graphviz[9], as it can easily be generated and then compiled in a variety of different styles, and, on top of this, it makes it easy to modify graphs manually, as well as separate the information extraction task from the rendering task. This means, however, that the python snippet will write graphviz code as output for these tasks - the print statements used for this should simply be ignored by the novice, reader, and is always wrapped in a standalone function explaining the contextual meaning of the generated graphviz code.

To do this analysis, the full capabilities of the Python interpreter will be utilized, by analyzing code and structure with reflection[3] at runtime. This is partially as a technical challenge, but also to explore the capabilities of code analysis done via reflection.

The sourcecode of the various objects will be analyzed, however this will also be done via reflection, making the solution a mixture of dynamic and static code analysis. The upside of this, is that there is a certainty that the findings are reflecting the actual running piece of software at runtime.

The paper is written in org-mode in Emacs, and the sourcecode can be found [TODO here]. Any reader that utilizes the Emacs editor can browse the sourcefiles, and can more easily play around with the snippets and intermediate results, however this is only for advanced users. The repository also contains the final snippets as stand alone scripts in a slightly modified

[8]Van Deursen et al. – Symphony - View-Driven Architecture ReconstructionFile

[9]`https://en.wikipedia.org/wiki/Graphviz`

version for readability, and can be inspected for any interested novice python user.

4 Data Gathering

Before delving into the generation of the views, we will start by gathering information about the Flask package. Initially we install the package from pip[10], whereafter we simply import the module, and check the docstring, the string containing the explanation of the package. Python objects has various so called *magic methods*,

```
import flask
print(flask.__doc__)
```

```

    flask
    ~~~~~

    A microframework based on Werkzeug.  It's extensively documented
    and follows best practice patterns.

    :copyright: 2010 Pallets
    :license: BSD-3-Clause
```

This lets us know that the framework should essentially be small in size, which is nice for the sake of this analysis, but it also mentions a primary dependency on the *Werkzeug* package. To figure out exactly what Werkzeug does, we can simply import that and read their docstring. This is possible because the installation of flask also installs all dependencies, which we then have access to as well.

```
import werkzeug
print(werkzeug.__doc__)
```

```

werkzeug
~~~~~~~~
```

[10] https://pypi.org/project/Flask/

```
Werkzeug is the Swiss Army knife of Python web development.

It provides useful classes and functions for any WSGI application to
make the life of a Python web developer much easier. All of the provided
classes are independent from each other so you can mix it with any other
library.

:copyright: 2007 Pallets
:license: BSD-3-Clause
```

This is less relevant for Flask, however it is interesting to know. We can hope to see that flask outsources some things, and that the sourcecode and structure is simpler. It is also furthers to understand the context of Flask.

For completeness, before looking into the gathering more information, it is crucial to understand how to use, from an end-user perspective, the framework. The following snippet is taken from their documentation.[11]

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

Next lets look at all objects and classes, that flask framework exposes, i.e all modules, functions, types and constants that can be imported by importing the flask framework. We will generate a helper function for fetching these, utilizing the `dir` function, which returns a list of attributes of a given object, here a unspecified module.[12]

```
def get_objs(module):
  return [
    (name,getattr(module,name))
    for name in dir(module)
  ]
```

Any module and instance We will ignore all private identifiers, which in python is denoted by a `_` prefix. We also want to know the type of each

[11] https://flask.palletsprojects.com/en/1.1.x/quickstart/#a-minimal-application

[12] https://docs.python.org/3.8/library/functions.html#dir

attribute, to understand what it is. The output list is shortened for the sake of readability, however the reader is encouraged to check all output if interested. We could also check the docstring, if relevant, however this is left for the user, if interested in specific attributes.

```
import flask

def print_list(elements):
  # cast to list, so it works for generators and sets
  elements = list(elements)
  formatted ='\n- ' +('\n- '.join(elements[:5]))
  print(
    f"{len(elements)} elements. "
    f"Among others:{formatted}"
  )

print_list(
  f"{label} ({type(obj)})"
  for label, obj in get_objs(flask)
  if not label.startswith("_")
)
```

```
55 elements. Among others:
- Blueprint (<class 'type'>)
- Config (<class 'type'>)
- Flask (<class 'type'>)
- Markup (<class 'type'>)
- Request (<class 'type'>)
```

We can then see that flask exposes a large set of different types, which are classes, and upon expecting the full list also functions, submodules, functions and classes. We can see some interesting types, forexample the `Request` class, which presumably is of big interest in a web framework, but also the `Flask` class, which is a crucial class when using the framework. It however, also shows that the structure is tree like, and not flat. This is obvious based on some of the submodules exposed, i.e `flask.testing`.

We can also, alternatively, look at all the modules loaded into the interpreter when loading in the `flask` module. We do this by first getting the list of all modules prior to import flask, and then taking the difference of that with the modules loaded after importing flask.

```
import sys
initial_modules = set(sys.modules)
import flask
diff_modules = set(sys.modules) - initial_modules
print(f"{len(initial_modules):>3} initial modules")
print(f"{len(diff_modules):>3} imported modules")
print(f"{len(sys.modules):>3} loaded modules")
```

```
 57 initial modules
205 imported modules
262 loaded modules
```

This points in the direction of the complexity of the system, as it shows all the external as well as internal modules. It does not say anything about the size of each of those modules, however by inspecting the sourcecode, we could generate this. For this we generate a `LOC` function that checks the total number of lines of code for a given module. We utilize the `inspect.getsource` function for this, however this only works on non-builtin objects. The LOC also includes comments and blank lines, and shouldn't be taken as a clear metric of the system, however it does give a ballpark estimate of the size of the project

```
import inspect
import sys
initial_modules = set(sys.modules)
import flask
diff_modules = sorted(set(sys.modules) - initial_modules)
def LOC(module):
  try:
    return len(inspect.getsource(module).split("\n"))
  except (OSError,TypeError):
    return 0
def sum_loc(iffunc = None):
  return sum(
    LOC(sys.modules[module])
    for module in diff_modules
    if not iffunc or iffunc(module)
  )
imported_lines = sum_loc()
flask_lines = sum_loc(
  lambda module: module.startswith("flask")
)
werkzeug_lines = sum_loc(
  lambda module: module.startswith("werkzeug")
)
pct_flask_size = flask_lines/imported_lines*100
pct_werkzeug_size = werkzeug_lines/imported_lines*100

print(f"{imported_lines:>5} imported lines of code")
print(f"{flask_lines:>5} lines of code in flask")
print(f"Flask covers {pct_flask_size:.2f}% of total lines")
print(f"Werkzeug covers {pct_werkzeug_size:.2f}% of total lines")
```

```
94841 imported lines of code
 7728 lines of code in flask
Flask covers 8.15% of total lines
Werkzeug covers 18.10% of total lines
```

This shows that the flask application is rather small compared to the total dependencies, and does point towards the statement made in their docs, that it is light weight, and that it is heavily dependent on external dependencies, especially the Werkzeug module.

5 Graph creation

We have gained an initial understanding of what Flask is, how it is used, and what submodules and types are being exposed. From here we can delve into the generation of the graphs, and look at a more structural graphical view of the system.

5.1 Component Diagram

To get a graph we need to know what is a direct dependency and what is an indirect dependency. To do this we will, once again, do static code analysis with the help of the `inspect` module. Here we can fetch any line starting with `import`. To do this we create a `get_imports` function.

```
import inspect
import re
```

```
def get_imports(module):
  regex = r"^(import|from)\s(\.?\w*)"
  source = inspect.getsource(module)
  return sorted(set(
    # get the ‘(\.?\w*)‘ pattern in the match
    match[1]
    for match in re.findall(regex, source, re.MULTILINE)
  ))

import flask
print_list(get_imports(flask))
```

```
13 elements. Among others:
- .
- ._compat
- .app
- .blueprints
- .config
```

This gives us a list of the direct imports, which is a lot more interesting, and can be used to generate a graph. We have to understand the imports however - anything prefixed with a dot, i.e `.app` is a relative import, so the full qualifier would be `flask.app`, because we are currently in the flask application. we can convert these to absolute name, and ignore the `.` import.

```
from typing import List
from types import ModuleType
def get_abs_imports(
    module:ModuleType,
    base_name:str
) -> List[str]:
  return [
    (
      base_name + imp
      if imp.startswith(".")
      else imp
    )
    for imp in get_imports(module)
    if imp not in ["","."]
  ]
```

```
print_list(get_abs_imports(flask,"flask"))

12 elements. Among others:
- flask._compat
- flask.app
- flask.blueprints
- flask.config
- flask.ctx
```

We can then use this list to import each module and then look at the source for each of these, and with that recursively check all dependencies. We then generate a graph based on this recursive process. The color of external dependencies are set to dark gray to easily differentiate between internal and external dependencies. The following snippet shows all the helper functions we will create for recursively mapping these dependencies.

```
import importlib
# to make sure we don’t get an infinite loop
visited = []

def print_dependency(start:ModuleType, end:ModuleType, color:str=""):
     color = f’fontcolor="{color}" color="{color}"’ if color else ""
     print(f’{hash(start)} -> {hash(end)} [{color}]’)
def print_label(module:ModuleType, color:str=""):
     color = f’fontcolor="{color}" color="{color}"’ if color else ""
     print(f’{hash(module)} [label="{module.__name__}" {color}]\n’)

def recursive_print_imports(
    module:ModuleType,
    base_name:str,
    only_internal: bool = False):
   if module in visited:
     return
   visited.append(module)
   for import_name in get_abs_imports(module, base_name):
     internal = import_name.startswith(base_name)
     if not internal and only_internal:
       continue
     import_module = importlib.import_module(import_name)
     color = "darkgrey" if not internal else ""
     print_dependency(module, import_module, color)
     print_label(import_module, color)
     if internal:
       recursive_print_imports(
         import_module,
         base_name,
         only_internal
       )

def chart_imports(module:ModuleType, only_internal:bool = True):
  print_label(module)
  recursive_print_imports(module, "flask", only_internal)
```

These helper functions can then be utilized to render the component diagram. Initially I only checked the `flask` module, however i later realized that it did not import the testing modules. Therefore the following snippet, which generates the component graph, also checks the `flask.testing` module. Initially, lets only look at the internal module.

```
# The previos snippet isnt rerun everytime we run this,
# seo we have to reset visited.
visited = []
import flask
import flask.testing
chart_imports(flask, only_internal=True)
chart_imports(flask.testing, only_internal=True)
```

"images/""dependency-internal".png

The structure seems clear and well though out. You can see a clear hierarchy, and a modular construction of the system. This is naturally a good sign. We also see that there is no circular dependencies, and, based on the titles of the modules, seems to have a clear separation of responsibilities.

```
# The previos snippet isnt rerun everytime we run this,
# seo we have to reset visited.
visited = []
import flask
import flask.testing
chart_imports(flask, only_internal=False)
chart_imports(flask.testing, only_internal=False)
```

"images/""dependency-external".png

Here we can see various dependencies, of different types and importance. At the lowest level, with the largest count of on incoming edges, we see the `Werkzeug` module. This highlights the fact that there is a closely knitted relationship to `Werkzeug`. The base module, `flask`, depends only on very few external dependencies, and with the external modules included, it is clear to see that functionality is wrapped through the sub modules.

From this we can see a clear structure, additionally the architecture seems to be of high quality.

5.2 Class diagram

The initial list of identifiers we fetched, via the ==

Looking at the attributes exposed in the initial data gathering snippet, we can see a both classes and instances. Anything of type `<class 'type'>` is a class, as the type of a class is a type, whereas any "specific" type, anything that isn't a python-keyword, is an instance, for instance `<class 'flask.signals._Fakesignal'>`. The list of classes and instances is however rather long, so initially I would like to look simply look at the `Flask` object, as that is the Application class, the main class of the framework.

Let's first check if it implements any super classes.

5.2.1 Inheritance

```
import flask
dependencies = [
  f"{dep.__name__}:\n  Docs: {dep.__doc__}"
  for dep in flask.Flask.__bases__
 ]
print("Flask is based on:",*dependencies, sep="\n")
```

```
Flask is based on:
_PackageBoundObject:
  Docs: None
```

We learned little other than it depends on an undocumented class.

Instead we can try to look at what different functions, classes and instances the `Flask` class references.

Worth noting is that due to the dynamically typing of python, it can be difficult to get a list of dependencies and references inside the function, however we can see any direct usage, however any expected input types of various functions aren't possible to deduce without any typing or usage examples.

5.2.2 Check for references to classes in sourcecode

As we can get the list of all defined classes, functions and variables accessible in the flask application, it should be relatively painless to check if these are defined inside the sourcecode of this class. We will limit our search to classes, and ignore dependencies on functions and modules, even though functions might be interesting as well, however that will be considered an alternative view. It is however important that we check the `__dir__` of the correct module. We can easily see where the Flask application is defined by simply checking the output of the `__str__` function by printing it:

```
import flask
print(flask.Flask)
```

```
<class 'flask.app.Flask'>
```

Which then means that we have to check `flask.app.__dir__`.
The following snippet checks the source code of

```
import inspect
import re
import flask


#remove comments
def remove_comments_and_strings(string):
    string = re.sub(re.compile("#.*?\n" ) ,"" ,string)
    string = re.sub(re.compile("\"\"\".*?\"\"\"", re.DOTALL) ,"" ,string)
    string = re.sub(re.compile("\".*?\"" ) ,"" ,string)
    string = re.sub(re.compile("\'.*?\'" ) ,"" ,string)
    string = re.sub(re.compile("\`.*?\`" ) ,"" ,string)
    return string

source = inspect.getsource(flask.Flask)
identifiers = flask.app.__dir__()
source = remove_comments_and_strings(source)
used_identifiers = [
  f"- {identifier}"
  for identifier in identifiers
  if re.findall(f"[^.\w\d]{identifier}", source, re.MULTILINE)
  and identifier is not flask.Flask.__name__
  and isinstance(getattr(flask.app, identifier), type)
]
print(
  f"count: {len(used_identifiers)}",
  *used_identifiers[:5],
  sep="\n"
)
```

```
count: 28
- timedelta
- chain
- Headers
- ImmutableDict
- BadRequest
```

We here see a lot of direct dependencies through usage, which also shows interesting classes in regards to the structure of flask itself. Presumably the `Request` class is interesting to look at, and look at the attributes of, when handling any data sent to the server, for example. We see some rather generic dependencies as well, for instance `timedelta`, which is a python class used for keeping track of relative time. We also see a couple of different exception based classes.

5.2.3 Generating the graph

Based on these two principles, we can generate a graph that looks at association as well as inheritance. The following snippet recursively checks all classes exposed by `flask`, and maps their references (these are both in attributes, calls and inheritance), and then maps all these throughout the framework, stopping when reaching an external dependency.

We ignore the methods and attributes of the classes and only focus on the relations between them. The current documentation is a great source for the content and intent of each class.

Any internal classes that are not linked to any of the base classes are not displayed as they are assumed to be ghost code. Grey arrows are representing association, and black arrows are representing inheritance. This is not UML, however it does, in my opinion, obviously show what is what, and is easier to see on the relatively clustered diagram.

In the following snippet i fought a lot with module imports. it is rather easy to detech a mention of a specific class, i.e `=Flask=`, however i soon realized that if the class usage was based on a module import, then the `__dir__` function naturally only showed the module, not all the classes that module exposed, and therefore wouldn't reveal ==

```
from types import ModuleType
import inspect
import re
import flask
import flask.testing

def escape(label):
  # Some identifier names aren't acceptable by graphviz
  return label.replace(".","__")

blacklist = []
```

```
def draw_dependency(klass, dep):
  color = '[color="darkgrey"]' if not dep in klass.__bases__ else ''
  dep_label = dep.__name__
  print(
    f'{hash(klass)} -> {hash(dep)} {color}\n'
    f'{hash(dep)} [label=\"{dep_label}\"]\n'
    f'subgraph cluster_{escape(dep.__module__)}{{\n'
    f'{hash(dep)};\n'
    f'label=\"{dep.__module__}\"\n}}'
  )

def map_dependencies(klass):
  if klass in blacklist or not klass.__module__.startswith("flask"):
    return
  blacklist.append(klass)
  source =inspect.getsource(klass)
  source = remove_comments_and_strings(source)
  module_name =klass.__module__
  # get the module by name
  module = eval(module_name)
  identifiers = module.__dir__()
  for identifier_name in identifiers:
    identifier =getattr(module, identifier_name)

    if identifier is klass:
      continue
    # if identifier is a module do something:
    if isinstance(identifier, ModuleType):
        matches = set(
          match[0]
          for match in
          re.findall(f"[^.\w\d]({identifier_name}(\.[\w\d_]+)+)", source, re.MULTILINE
          )
        for match in matches:
            try:
              while match and not isinstance(eval(module_name+"."+match), type):
               match = ".".join(match.split(".")[:-1])
            except AttributeError:
              continue
            if not match:
              continue
            draw_dependency(klass, eval(module_name+"."+match))
    elif (isinstance(identifier, type)
      and re.findall(f"[^.\w\d]{identifier_name}", source, re.MULTILINE)):
        draw_dependency(klass, identifier)
        map_dependencies(identifier)

for identifier_name in flask.__dir__():
  identifier = getattr(flask, identifier_name)
  if isinstance(identifier, type):
    print(f'{hash(identifier)} [label=\"{identifier_name}\"]\n')
    map_dependencies(identifier)
for identifier_name in flask.testing.__dir__():
  identifier = getattr(flask.testing, identifier_name)
  if isinstance(identifier, type):
    print(f'{hash(identifier)} [label=\"{identifier_name}\"]\n')
    map_dependencies(identifier)
```

`"images/""class-diagram".png`

This diagram shows a lot of dependencies of the `Flask` class. This is expected as Flask essentially serves as the entry point for the whole framework, and all interactions with the WSGI system. This does, however, make it difficult to change the dependency with `Werkzeug`. Whether this is a problem, is probably a matter of personal taste.

6 Evaluation

Without doing proper acceptance tests it is difficult to qualitatively evaluate the snippets developed. However I, to mimic a maintenance task, looked at an issue with the testing framework, *Unable to extend FlaskClient with* $follow_{redirects}$,[13] and was able to get an overview of the files at fault. It did show some shortcomings of the current format. Normally an architectural

[13] `https://github.com/pallets/flask/issues/3396`

document is currated somewhat and via human interaction is pruned to only contain the relevant classes. Finding something in the current solution is rather difficult for this reason. The class diagram, for instance, is rather large and it took me some time to find the classes in question, however after finding it, it did help me understand the context of the problem. I looked at the `open` method of the `FlaskClient`, and already knew what it was inherited from and the context.

When doing this exercise I did consider whether this reflection-based code analysis was preferred, however I did see multiple upsides to this model. These are mainly related to the fact the python is an intepreted language, meaning that static code analysis can lead to various errors. one example of the problems of static code analysis was when I browsed the `flask` source-code on Github[4], where Github can show you references to a class you are looking at, however the below image shows an error of this function, as i am looking at the `EnvironBuilder` of the `werkzeug` module, however the references are upon closer inspection, actually references to `flask.testing.EnvironBuilder`, not `werkzeug.test.EnvironBuilder`, however this is difficult to know based on the level of static code analysis.

Github Static Code Analysis fails

The example is just a random bug, however it still shows an interesting aspect of reflection. Sometimes, especially regarding dynamically typed languages, reflection gives a more truthful answer than static code analysis, as you can check exactly what a given variable actually is, rather than assume based on the identifier name.

7 Conclusion

The two graphs show the structure of flask as well as how it dependens on it's various dependencies.

We can see a close knitted relationship with `werkzeug`, which is expected based on the description of the `Flask` framework. We can see that the system seems to have a clear structure, and that only a singular cyclic dependencies exists. We can also see that the sourcecode is relatively small and simple, based on the class count.

On top of me gaining an interesting insight in the inner workings of the flask application, without reading the source code line for line, we can see, through the evaluation that the graph can be utilized to gain an understanding of the system to a degree that issues are in fact easier to solve. This points to the fact that graphs and an overview of this type does help collaboration, and might encourage more to help maintain open source projects of various types. The current state of the scripts can be used to generate more specialized scripts for a given project at hand, or a more generic tool that can generate architectural diagrams for any open source system.

I would argue that the work is a partial success however, both in the fact that it brings light on the structure of the flask application, as well as the value of the scripts as a stand-alone achievement.

8 Future Work

8.1 Interactive tool

It would definitely help if the graph were more interactive. Looking through the diagram was difficult, and could use a search function of sorts. Additionally it would be nice to be able to connect the graph to the documentation. A feature could be the ability to click on a class on the diagram, and that bringing up the source code. This would however require an interactive tool of sorts, and was outside the scope of this paper.

8.2 Historical data extraction

By generating these graphs as code it can be easy to generate it for various versions of the sourcecode, via version control, and with that generate an evolutionary graph, essentially highlighting the difference between two graphs and showing any additions and deletions. A small animation could show this evolution, but also show changes that result from pull requests, for instance. This could be usable as a crucial step in a code review phase, where a similar graph could show the structural changes a given pull request creates, which is as interesting, if not more, than the difference in code.

8.3 Different graphs

Through other types of code analysis it would be interesting to see, if possible, the generation of various scenarios and usage of the system. You could also, through version control, highlight what areas of the code is most contested and most frequently causes conflicts, for instance. The two graphs shown here are relatively simple, and even more knowledge could be extracted with more time and work.