Architectural reconstruction

Introduction

This report will do a software reconstruction after the Symphony process. It will dissect the Rails framework and attempt to make logical interpretations on the architecture, functionality and complexity. Symphony is a software architecture reconstruction process that is heavily based on views. It is split into five steps:

1. Problem elicitation
2. Concept determination
3. Data gathering
4. Knowledge inference
5. Information interpretation

First the problem, that wants to be achieved, needs to be defined. After this, the information and views that might solve this problem need to be defined. When this is done, this data is being gathered and put into the views previously defined. With the abstracted information an interpretation can, be performed and hopefully a solution found.

Problem elicitation

When following an education in a computer science relevant field, students will be taught the fundamentals of computers and programming. This involves many topics as it is a complex topic. What an education most of the time does not include—for good reasons, that shall not be argued here, as this is not the focus of the report—is the work with frameworks. Frameworks are a set of reusable libraries for software systems. A popular area of computing where the number of frameworks is countless is for the development of web applications. These web frameworks have been built most of the time by a single person and then evolved to a code base, so complex that many developers actively maintain and expand it. Due to the complexity of these frameworks one focus point has been to make the initial set up and usage as easy as possible, often being able to have a running web application within minutes. This is valuable but comes with the risk of using the framework without knowing what is actually happening. Documentation is a key aspect of enabling this bridge from practice to theory and vice versa. Another part of open-source frameworks is that they rely heavily on the community to chip in, as open-source lives from contributions from everyone. But how does one contribute to a code base that has 394,884 lines of code, increasing by the hour. Documentation helps here as well, but is harder to maintain, because it would have been updated with every code change, whereas the documentation of the frameworks' functionality only needs updating when a major feature gets added or updated. This report aims at giving a programmatic overview of the Rails framework. It will extract data straight from the repository and map it to useful statistics and graphs. Further it will outline some evolutionary analytics to see where the most changes within

the project are occurring over time.

Concept determination

Before gathering the data from the codebase, it is important to define the architectural information that is needed to solve the stated problem of gaining a better overview and understanding of Rails. It is also beneficial to set the viewpoints that are planned to use to show the gathered information.

From previous studies of Rails, reading the documentation or looking at the root directory of the repository, it is clear that there are several components that form Rails. Good explanations what those individual components do, can be found in the official Rails documentation. Other information that would be useful that is not stated in the documentation:

- How complex are these components?
- How are the components connected to each other?
- How active is the development on them?
- What components experience change at the same time?

The concrete information that should be gathered is something like:

- Number of files in component
- Lines of code per file, per component
- Functions per file
- Function length
- External dependencies per component
- Usage between Rails components
- Commits with their modification to files/component
- Modification complexity
- Logical coupling of the components

Useful viewpoints to make sense of the data would be:

- A simple table showing the lines of code per file and then per component
- A bar graph showing the total lines of code in a component
- A node, edge graph showing the dependency between components
- The same graphs can be used to show modifications on files over time

Data gathering

In order to gather relevant information to answer the list of questions and ultimately solve the above stated problem, the Python language will be used with a few useful libraries to make the extraction of data easier. For the problem to solve two different sources of data will be used. Firstly, the actual source code of Rails will be used and analyzed, secondly the Git history will be looked at. The source code for Rails is hosted on GitHub and can be easily cloned. With a clone the source code will be made available locally and additionally a hidden folder `.git` is created and populated, which holds all Git relevant data—also

the history, that will used. Therefore, a clone of the repository makes all data needed available.

```
$ git clone git@github.com:rails/rails.git
```

Reconstruction

As before mentioned all components are in the root directory of the repository. The first step to see what components are available and load them into memory:

```
directories = []
for (dirpath, dirnames, filenames) in walk('/path/to/rails'):
    directories.extend(dirnames)
    break
# filter away all hidden directories
directories = list(filter(lambda x: x[0] != ".", directories))
```

A look inside one of those directories will give an overview of what a component actually is. There has been a standard format established for the development of small libraries in the Ruby programming language. These libraries are called *gems* and most often contain the following files:

- `lib/`: contains the source code
- `test/` or `spec/`: for the test files
- `Rakefile`: uses Rake to automate tasks, like testing, generating code
- `bin/`: includes executables and is loaded into the user's `PATH`
- `README.md`: for a descriptive text about the gem
- `*.gemspec`: holding general information, like author, version, external dependencies etc.

What is a gem? – Guides RubyGems

The directories give a good way to group and separate the information gathered by gem, as the information extraction can just be done on the directories after each other and stored in a suitable data structure and as they are all in the same format adds to the convenience.

Rails components:

- `actioncable`
- `actionmailbox`
- `actionmailer`
- `actionpack`
- `actiontext`
- `actionview`
- `activejob`
- `activemodel`
- `activerecord`
- `activestorage`
- `activesupport`

- railties

The project does not only include the Rails components but also other directories:

- ci
- guides
- tasks
- tools

These will be iterated, and every file will be looked at.

```
def get_files(path, file_extension):
    files = Path(path).rglob("*." + file_extension)
    meta_data_files = {}
    for file in files:
        meta_data_files[str(file)] = {
            'filename': str(file),
            'no_lines': number_of_lines(file),
            'no_functions': number_of_functions(file),
            'no_modules': number_of_modules(file),
            'no_requires': len(extract_require(file)),
            'functions': extract_functions(file),
            'requires': extract_require(file),
            'namespaces': extract_namespace(file),
            'autoloads': extract_autoload(file)
        }
    return meta_data_files
```

This loop will be called on the project root and given the Ruby file extension `rb`, to only look at Ruby files. This will count the number of lines, functions, modules, requires. Modules can be counted by extracting on the key word `module`, it is used in Ruby to indicate a files namespace, which will then be used to import it into a different file. It is useful for grouping code that belongs together.

```
module Mathematics
  module NumberTheory
    module Arithmetic
      def self.add(a, b)
        return a + b
      end
    end
  end
  module Geometry
  end
end

# Mathematics.NumberTheory.Arithmetic.add(1, 2) => 3
```

Using the `require` function is a way of importing files into the file that it is being called from. This is a good way of checking how many dependencies a file has to other files—showing possible complexity. There is one problem though: Rails, does not use this way of importing its own dependencies. What Rails does is, it uses an `autoload` function, that finds it in the path and loads it into the context of the library by having the class/file name passed to it.

```
module ActionView
  # …
  autoload :Base # loads the file base.rb
  # …
end
```

This can be extracted and used to show what kind of files are being loaded on startup of the library and needed *globally* for the library.

Further, the functions will be extracted. This list will hold a tuple of the function name and the line count.

A key function in the retrieval is this `extract_from_line` function. It will get a key word that it supposed to find on the provided line. Regular expressions are used to find the key word and the wanted data, like the function name.

```
def extract_from_line(name, line):
    if re.search("^([\s]*#)", line): # ignore comments
        return None
    elif name == 'def|end':
        method = re.search("^([\s]*(def |end)[ (\S+)]*)", line)
        return str(method.group(1)) if method else None
    elif name == 'module|class|end':
        namespace = re.search("^([\s]*(module|class|end)[ (\S+)]*)", line)
        return str(namespace.group(1)) if namespace else None
    else:
        x = re.search("" + name + " (\S+)", line)
        return None if x == None else str(x.group(1))
```

Because Ruby's syntax does not use any symbols indicating a block, but uses the off-side rule, just like Python, it is a bit trickier to extract functions and therefore, the code to do that is provided in the following listing.

```
def extract_functions(file):
    functions = []
    line_count = 1
    current_function_name = ''
    def_identation = -1
    for ext in extract(file, 'def|end', True):
        # If ext is None, it means it did not find a nested def or
        # an ending to the function, thus incrementing
        if ext == None:
```

```
            line_count += 1
        else:
            identation = len(ext) - len(ext.lstrip(' '))
            # Single line function
            if 'def ' in ext and ' end' in ext:
                current_function_name = retrieve_function_name(ext)
                functions.append(add_function(current_function_name, 1))
            # Beginning of new function
            elif 'def ' in ext:
                def_identation = identation
                line_count = 0
                current_function_name = retrieve_function_name(ext)
            # Function ending
            elif 'end' in ext and identation == def_identation:
                functions.append(add_function(current_function_name, line_count))
                current_function_name = ''
                line_count = 1
            else:
                line_count += 1
    return functions
```

This function extracts every line from a file and checks what key word is present, based on that it will either start a new function, end one or just increment its line count.

The initial `get_files` function is being called when instantiating the `rails_components` object.

```
rails_components = {}
for directory in directories:
    files = get_files(full_path('/' + directory + '/'), 'rb')
    average_LOC = int(
        reduce_by_key(files.values(), 'no_lines') / len(files.values())
    )
    average_NOF = int(
        reduce_by_key(files.values(), 'no_functions') / len(files.values())
    )
    average_requires = int(
        reduce_by_key(files.values(), 'no_requires') / len(files.values())
    )
    dependencies = get_external_dependencies(directory)
    rails_components[directory] = {
        'files': files,
        'average_LOC': average_LOC,
        'average_NOF': average_NOF,
        'average_requires': average_requires,
        'dependencies': dependencies }
```

```
rails_components = {
    k: v for k, v in sorted(
        rails_components.items(), key = lambda item: item[0]
    )
}
```

This will also add some general information, based on the gathered information about each individual component. Especially `dependencies` is of interested. The `.gemspec` file holds all the external gems that the current library depends on. This can be used to see how large the external complexity for the specific library is.

Evolutionary

`PyDriller` exposes a class called `RepositoryMining`, that comes in handy when analysing a repository project.

```
@functools.lru_cache(maxsize=None) # Memoize result for inputs
def count_file_modifications_simple(path, tag):
    commit_counts = defaultdict(int)
    for commit in RepositoryMining(path, from_tag=tag).traverse_commits():
        for modification in commit.modifications:
            try:
                commit_counts[modification.new_path] += 1
            except:
                pass
    return commit_counts
```

This function `count_file_modifications_simple` will count occurrences of a file in the commit history, giving a good overview of the number of times a file has been changed over its history. This implementation has the problem that it does not differentiate between, pure changes, deletion or additions of files. In order to handle these cases, the modification object gives access to: `old_path` and `new_path`.

```
# cc = {
#   'filename': occurrences in commit history,
#   'actionmailer/lib/action_mailer.rb': 113
# }
def handle_change_type(cc, commit, modification):
    new_path = modification.new_path
    old_path = modification.old_path
    try:
        # Change old_path to new_path, keep values
        if modification.change_type == ModificationType.RENAME:
            cc_old = cc.get(old_path, (0, []))
            cc[new_path] =  (cc_old[0] + 1, cc_old[1])
            cc.pop(old_path)
```

```
        elif modification.change_type == ModificationType.DELETE:
            cc.pop(old_path, '')
        elif modification.change_type == ModificationType.ADD:
            cc[new_path] = (1, [])
        else: # Modification to existing file
            count, cx = cc[old_path]
            comp = modification.complexity
            cc[old_path] = (count + 1, cx + [comp] if comp else cx)
    except Exception as e:
        pass
```

Another interesting metric is the complexity of a modification. This is also recorded on most of the modifications and is realized as the cyclomatic complexity.

Logical coupling can be used to detect modules that may depend on each other based on their frequency of change in the same commit. One can assume when two Rails components appear in the commit history often together, they depend on each other. This is of course no true indication like an actual dependency graph would give, but still valuable as it could give indications on dependencies that are not shown on dependency graphs generated from static analysis.

To couple the individual Rails components, firstly the components have to defined. This can be done manually as they do not change frequently and can be easily adapted when something gets added or removed.

```
def rails_components():
    return [
        'activerecord', 'activesupport',
        'actionpack', 'railties',
        'actionview', 'activemodel',
        'activestorage', 'activejob',
        'actionmailbox', 'actioncable',
        'actiontext', 'actionmailer'
    ]
```

The original function that counts the change occurrences of files can be reused, because it already loops over every commit and modification, so no reason to do it again.

```
rails_components_dictionary = rails_components_dict()

# Returns the Rails component from the path
# actionmailer/lib/action_mailer.rb -> actionmailer
def get_component_from_modification(modification):
  return modification.old_path.rsplit('/')[0] or modification.new_path.rsplit('/')[0]

# cc = {
#   'filename': (occurrences in commit history, complexities),
```

```
#   'actionmailer/lib/action_mailer.rb': (113, [1,1,1,…]
# }
def logical_coupling(path):
    cc = defaultdict(lambda: (0, []))
    for commit in RepositoryMining(path).traverse_commits():
        # Holds all components that were touched in commit
        changed_components = []
        for modification in commit.modifications:
            # Holds changed component
            comp = get_component_from_modification(modification)
            # Check if it is a Rails component change
            if comp in rails_components():
                changed_components.append(comp)
            # Was the try-block in the previous version
            handle_change_type(cc, commit, modification)
        # Adds all changed components, but itself to the dictionary
        add_other_components(rails_components_dictionary, changed_components)
        # Reset the changed components for each commit
        changed_components = []

    return cc
```

All the information gathered in both processes will be plotted either by using basic `print` in a tabulated format or plotted by using `networkx` and `matplotlib`.

Knowledge inference

In this step the low-level information gathered shall be abstracted to ease its readability. This will be done by mapping the information to the selected views.

Printing some initial statistics about the Rails components with the `print` and `format` function.

```
def print_rails_components():
    format = "{:<15}{:<1}{:<10}{:<1}{:<7}{:<1}{:<11}
              {:<1}{:<14}{:<1}{:<10}{:<1}{:<10}"
    print(format.format(
        "Component", " | ", "Ruby files", " | ",
        "Ø LOC", " | ", "Ø Functions", " | ",
        "Ø Function LOC", " | ", "Ø Requires",
        " | ", "Dependencies"
    ))
    print('---------------+-----------+---------+
        ------------+---------------+-----------+-------------'
    )
    for k, v in rails_components.items():
        print(format.format(k, ' | ' , len(v['files']), ' | ',
```

```
        v['average_LOC'], ' | ', v['average_NOF'], ' | ',
        v['average_function_LOC'], ' | ', v['average_requires'],
        ' | ', len(v['dependencies'])
    ))
```

Component	Ruby files	Ø LOC	Ø Functions	Ø Function LOC	Ø Requires	Dependencies
activerecord	853	160	12	3	1	2
activesupport	469	119	10	4	1	5
actionpack	325	202	16	4	1	6
railties	278	148	9	4	3	5
actionview	186	231	18	3	1	5
activemodel	130	118	8	4	1	1
activestorage	124	62	3	1	1	5
activejob	118	70	4	2	1	2
actionmailbox	93	34	1	2	0	6
actioncable	86	79	6	3	1	4
actiontext	74	35	2	2	0	5
actionmailer	41	140	9	3	1	6

Figure 1: All Rails components broken down by file

To have an even clearer visual representation of the distribution of the number of files in Rails a pie chart can be used.

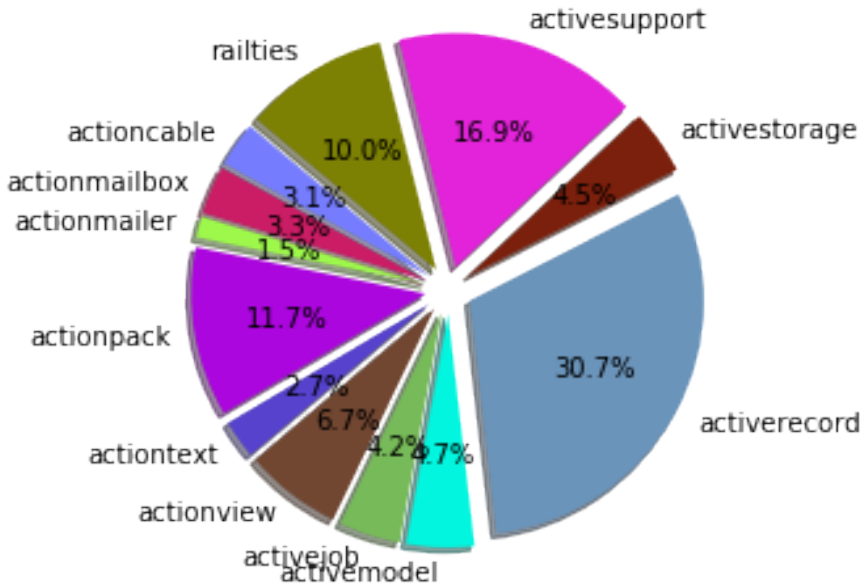

Figure 2: Pie chart distribution by number of Ruby files

Component	LOC	NOF
activerecord	136860	10949
actionpack	65919	5289
activesupport	56011	4714
actionview	43045	3474
railties	41328	2653
activemodel	15441	1071
activejob	8265	559
activestorage	7710	389
actioncable	6846	535
actionmailer	5742	369
actionmailbox	3170	144
actiontext	2625	174

Figure 3: All Rails directories broken down by their total LOC and NOF

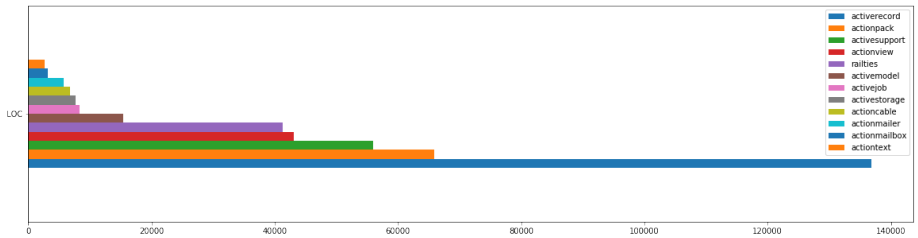

Figure 4: Bar distribution of the LOC by Rails component

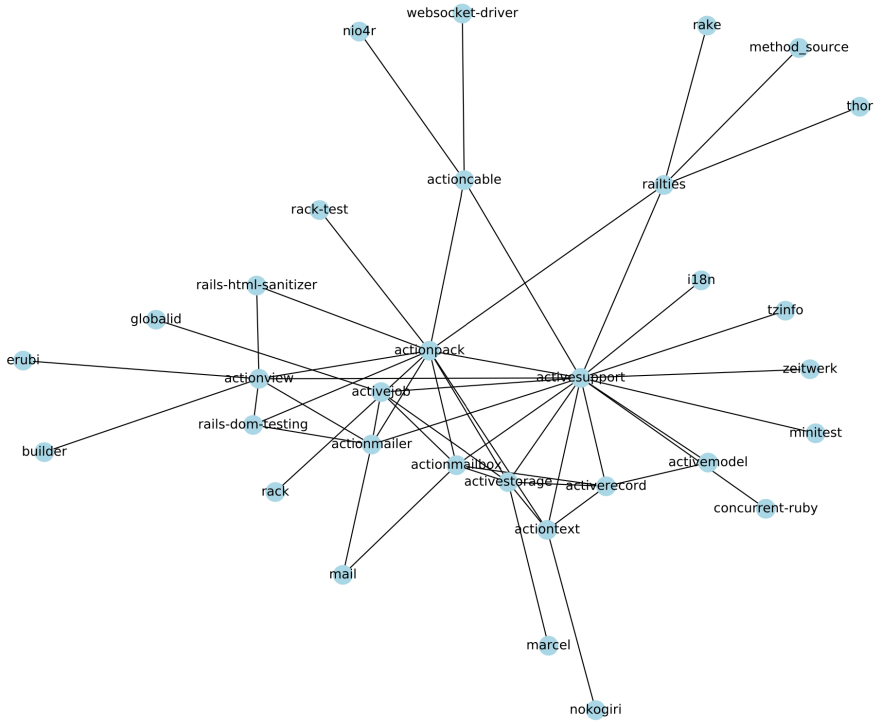

Figure 5: Rails component dependencies

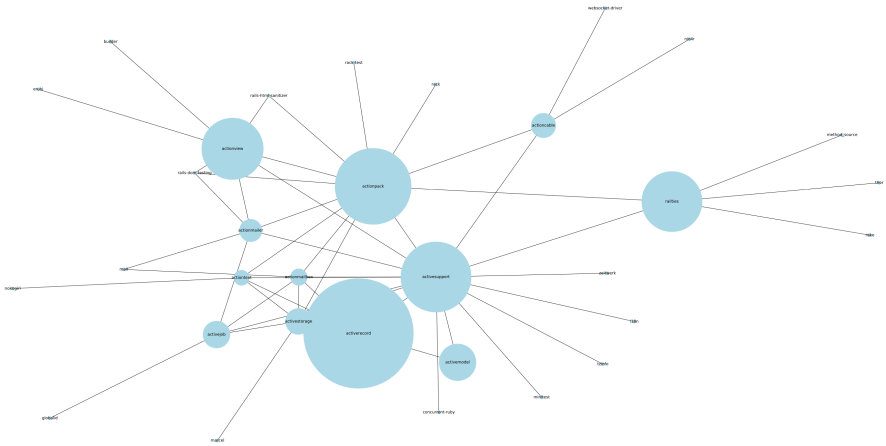

Figure 6: Rails component dependencies scaled by their LOC

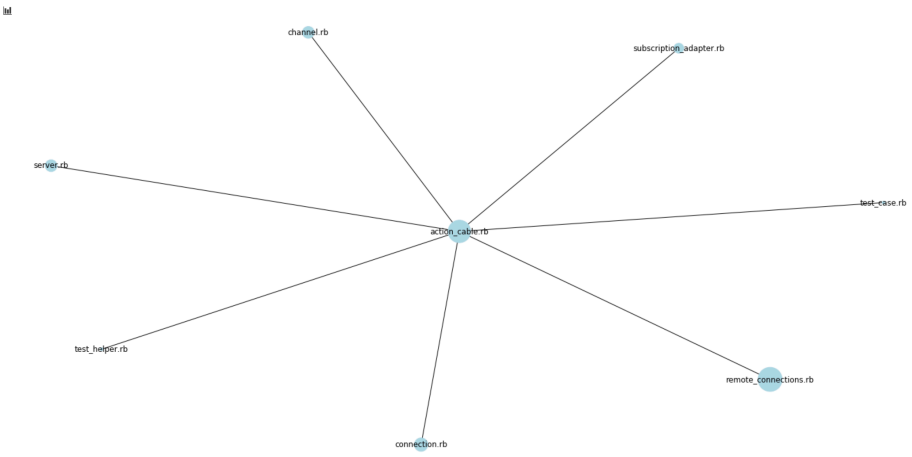

Figure 7: Libraries that are being loaded for `actioncable` *on startup, nodes scaled by LOC*

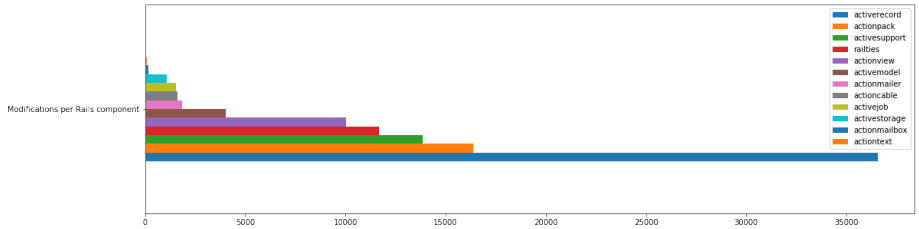

Figure 8: Total file changes recorded in the entire git history of Rails per component

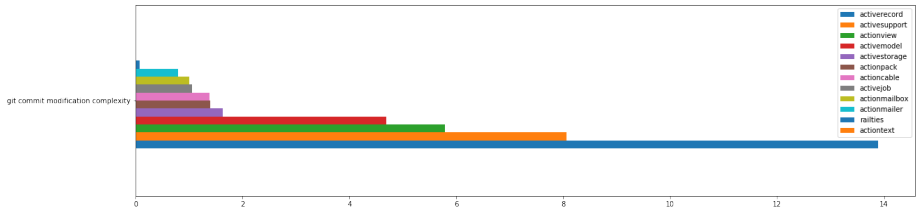

*Figure 9: Average modification complexity recorded in the entire git history of Rails per component**

component	actioncable	actionmailbox	actionmailer	actionpack	actiontext	actionview	activejob	activemodel	activerecord	activestorage	activesupport	railties
actioncable	x	60	7081	51645	76	31747	16632	19285	**112008**	210	70245	42721
actionmailbox	60	x	47	113	206	77	29	30	130	111	50	**767**
actionmailer	7081	47	x	42215	41	18134	9292	12764	**72305**	108	44069	30022
actionpack	51645	113	42215	x	134	167056	67057	89538	**512923**	514	318181	202884
actiontext	76	206	41	134	x	87	32	38	138	149	71	**607**
actionview	31747	77	18134	167056	87	x	40970	50138	**289649**	363	178098	105586
activejob	16632	29	9292	67057	32	40970	x	25068	**145045**	166	90583	53842
activemodel	19285	30	12764	89538	38	50138	25068	x	**186977**	258	114503	67524
activerecord	112008	130	72305	**512923**	138	289649	145045	186977	x	1113	650434	395371
activestorage	210	111	108	514	149	363	166	258	**1113**	x	480	796
activesupport	70245	50	44069	318181	71	178098	90583	114503	**650434**	480	x	244547
railties	42721	767	30022	202884	607	105586	53842	67524	**395371**	796	244547	x

Figure 10: File modifications from component that occurred in the same commit

Information interpretation

In this step the information and the views shall be interpreted and tried to make sense of with the goal of solving the stated problem.

In figure 1 and figure 3 one can observe that `activerecord` is by far the largest component in the framework, making it double the size of the second largest component by lines of code and number of functions. What is interesting is, that even though it is much larger, the relative sizes, like number of lines per file or per function, the number of requires is pretty much stable across all components. This is an indication for good design, as even though complexity arises it should not bring large and cluttered files with it. This indicates probably the concept of single responsibility in classes and functions. Another interesting part about the extracted statistics is the low number of imports by using the `require` function. Upon further investigation in the codebase and as already mentioned the `autoload` function is being used a lot. This explains the lack of requires in the files, because most classes are being loaded in the point of entry file of each component. This fact and the figure 7 show-casing the classes being loaded can be used to get a brief overview of what the component might do based on the naming of those classes. Figure 6 is one graph that was picked from all Rails components to show the idea. It shows seven dependencies that are being loaded in the beginning. Names that stand out and help to derive a sense of functionality are: `server.rb`, `remote_connections.rb`, `connection.rb`, `channel.rb`. This leads to the conclusion of it having to do with connections from client and server, also something with channels, this is most often used in WebSockets, where client subscribe to channels and data then travels bidirectional between client and server.

Looking at figure 5 and 6 shows the loaded external dependencies for each component. Edges are only drawn from Rails components to either external libraries or other Rails components. The external libraries' dependencies are not considered, nor their complexity by lines of source code. A striking fact is that `activesupport` has the most edges to other Rails components. This can be concluded as being a library with a lot of *supporting* code. A glance over an

excerpt of the autoloaded classes in `activesupport` confirms this.

- `Gzip`
- `JSON`
- `Benchmarkable`
- `Cache`
- `ProxyObject`
- `KeyGenerator`

The information from the evolutionary process only supports the previous made statements about complexity in `activerecord`. Figure 8 and 9 both have it as the the most active component by far. Figure 10 also shows that `activerecord` has its share in all components or that it is at least changed a lot. Interesting is that `actiontext` and `actionmailbox` do not have their strongest coupling to `activerecord`, which hints at not having any relations to the database helper component and being independet from it.

Conclusion

The reconstruction of Rails framework using the Symphony approach has been more or less successful. Doing the reconstruction with the purpose of gaining insights without working with stakeholders or system experts was not completely Symphony conform but did not block the process of the reconstruction. The reconstruction was though somewhat blocked by not being able to extract classes from the files in the Rails project. The reason for not being successful in this regard is that it was not trivial to extract the correct namespaces and classes of a file, as Ruby uses the off-side rule and modules are only used for *namespacing* code and the class key word then actual indicates if the file contains a class. If that would have been successful the next challenge would have been to find the dependencies of the classes to others, usually Ruby uses `require` to load files. Rails on the other hand has its own dependency loader, which uses another function from Ruby's kernel, called `autoload`, which has multiple implementations to support eager loading for example. This `autoload` function is smart enough to find the correct file, by only giving it the class name. The usage of this class then would need to be tracked in all files, in order to construct a class diagram. This is very unfortunate, and a lot of time was spent to first understand this and then find a solution. The final solution to use a dependency graph between the different Rails components, based on their loading from the `.gemspec` file was still satisfiable to some degree. The knowledge acquired through the static analysis of the complexity of the different components yielded great results as it shows, which of the components are complex and experience a lot of work, based on the number of modifications on them. The modifications found from the evaluation in the evolutionary analysis of the git history was also valuable. Firstly, as just mentioned, the visible number of modifications on the components, but also their average complexity let to believe that some components are more complicated as others. Secondly figuring out through logical coupling that some components change rather often together shows how tightly the framework

is coupled together, even though it might not seem in the beginning from the lack of dependency on each other.