Codoc: Code-driven Architectural View Specification Framework in Python

Casper Weiss Bang
IT University of Copenhagen
Copenhagen, Denmark
Email: c@cwb.dk

Mircea Lungu
IT University of Copenhagen
Copenhagen, Denmark
Email: mlun@itu.dk

Abstract—**Architectural views are expensive to maintain. Software systems continuously evolve, making architectural views outdated. We present Codoc, a continuous architectural documentation system. Codoc empowers users to utilize a code-driven architectural view specification language to create views that are continuously updated and accessible. Through qualitative studies, we find code driven view specification as a promising candidate to replace manual architectural visualization in software documentation.**

***Index Terms*—Architectural Views, Continuous Documentation,**
Participatory Design

I. Motivation

Software documentation is often outdated [1] [2]. The majority of agile developers find the level of internal documentation as *too little*, while at the same time assessing that documentation is either *important* or *very important* [3, p 161]. Leotta et al. [4] found that misaligned documentation can impact programmer productivity. While documentation is needed, there is also a motivation to *"Travel Light"* [5], i.e. minimize the amount of artifacts that needs to be maintained.

Software documentation often includes diagrams that visualize aspects of the underlying software. We use the term *architectural views*, meaning *"[...] representations of the overall architecture that are meaningful to one or more stakeholders in the system"*[1].

Osterweil [6] proposed that software development can be represented by software processes. He argued that *software practitioners* are used to define processes as code, and that the development of software, and the related practices, should be possible to represent as code too.

In this paper we investigate the possibility of specifying software visualization, and in particular, architectural views as code. We propose they are specified in a similar way in which developers specify tests with the help of frameworks like JUnit. We hypothesize that architectural-views-as-code might be more easily accepted by developers if they are easy to integrate in their *continuous**-practices [7]. Besides the ease of adoption, an added benefit is that the architectural views are less likely to become outdated because they can be define in such a way as to automatically change when the code changes. Furthermore, if the views are defined in code, developers can be notified when changes to the code affect a view and verify that it is still relevant.

Similar to testing, we believe that (architectural view) documentation should be a continuous practice that should part of the continuous integration workflows in such a way as the architectural views to alwasy be up to date.

Broadly, we aim to answer the following question: *How do developers react to a code-driven continuous documentation tool, which allows the definition of architectural views in the same language as the system?*

To answer this question we implement a code-driven code visualization tool for Python[2] that enables a user to specify which parts of a given software system they want to visualize in *architectural views*. Users subsequently test the tool. The tool handles the layout and visualization automatically (in a most basic way) and publishes the view in an online repository where stakeholders can access them.

II. Related work

A recent survey of the software visualization domain found no tools utilizing a code-driven interface. Instead most of them are driven by interactive user interfaces [8]. One exception that we are aware of is the Glamorous Toolkit[3] for Smalltalk in which views are defined in Smalltalk code. There are, unfortunately, few developers able to write Smalltalk at the moment.

Architecture Description Languages (ADLs) are used to formalize, via code, a given software architecture [9]. Languages like *ArchJava*, an extension to Java, enable the developer to embed *"[...] architectural features and enforce communication integrity*" [10]. However, in general, ADLs are not built to extract architectures, but rather formalize and visualize a planned architecture. And the reality is that few systems are nowadays designed with an upfront architecture specification.

Mens et al. propose the idea of *intensional views*, which utilizes metaprogramming combined with logic programming, to impose constraints and present various attributes of an underlying software system [11], [12]. Their research utilizes a rule-based system, making it easy for developers to see if a

[1]https://pubs.opengroup.org/architecture/togaf8-doc/arch/chap31.html

[2]One of the most popular languages at the moment https://insights.stackoverflow.com/survey/2020

[3]https://gtoolkit.com/

given rule is followed in a codebase. Their studies, however, do not address documentation or visualization explicitly.

Commercial tools like *Understand* [4] and Sourcetrail[5] are explorative and any exporting and filtering is done manually in a graphical user interface. Similarly, some IDEs also include features that will render diagrams, however, they are also explorative.

Finally, command-line-driven, non-interactive visualizers like the Python-specific *pydeps*[6] – have a limited capacity of view customization via command-line arguments.

III. Solution

We created a *Minimal Viable Product* (MVP) that exposes a code-driven interface for generating architectural views directly from source code and we tested it with users. The developed system is called *Codoc*, a name that stems from the abbreviation of *COntinuous DOCumentation*. The code is open source and available on GitHub[7] under the GNU General Public License. In this section we present version 1.0.0 of the tool, as published in the corresponding release on GitHub and through PIP.

This section illustrates with a step by step example how a developer named Alice can document a Python project called `sample` and share the created views with her colleague Bob.

Step #1. Getting an Online Account. To use Codoc, Alice needs a user account to access the online web application (the *Codoc web app*) and a *project*, which is a grouping of users and views.

Step #2. Installing the codoc Python package. Alice consults the Getting Started[8] guide in the documentation.

She installs the `codoc-python` package into the development environment by running the following in a shell:

```
pip3 install codoc-python
```

This installs both the view generation framework (the `codoc` Python package) and the command-line interface (CLI) for publishing views to the online repository of views hosted at codoc.org.

Step #3. Setting up the views folder. In a similar manner to having a special folder for unit tests, Alice creates a folder for all their architectural views in the project's root directory. The created folder has to be called `codoc_views`.

Inside the folder she creates a config file, named `config.py`. The config file is shown in Snippet 1, where `sample` is the name of the project that Alice wants to document.

```
1  from codoc import new_graph
2
3  import sample
4
5  # `**kwargs` are used to pass flags
6  # from the CLI to the tool.
7  def setup(**kwargs):
8          return new_graph(sample, **kwargs)
```

Snippet 1: A minimal configuration file

The lines that are specific to Alices project are #3 and #8. With them Alice specifies that she wants to create a *source graph* from the code in the "sample" package. The source graph is built by analyzing the source code in the sample package and consists of modules, classes, functions, and relationships between them[9].

Step #4. Defining a View. After copying it from the *Codoc* documentation, Alice pastes an example view into a new file named `top_level_modules.py`. The file is shown in Snippet 2 where several lines illustrate the API of *Codoc*:

L#3 the `@view` annotation signals that this function is to be treated as a view definition

L#6 the `modules` function takes as input a parameter named `graph` – that is the source graph that was created in the `config.py` above.

L#13 the `include_only_modules` filter removes from the graph classes and functions

L#14 the `depth_filter` function simplifies the graph further by only keeping the top-level modules

```
1   from codoc import filters, view
2
3   @view(
4   label="Top Level Modules",
5   )
6   def modules(graph):
7       """
8       The top level modules in our dear Sample project.
9
10      Also includes any direct descendants of
11      top level modules.
12      """
13      module_graph = filters.include_only_modules(graph)
14      depth_filter = filters.get_depth_based_filter(2)
15      top_module_graph = depth_filter(module_graph)
16      return top_module_graph
```

Snippet 2: A view function that generates views containing the top level modules of a given system

Step #5. Publishing a View. Having defined her first view Alice consults the documentation, which tells her how to publish her views (i.e., export an API key, and run `codocpy publish` in the terminal). The input and output of her terminal is shown in Snippet 3.

```
$ export CODOC_API_KEY="f35c0e821b8c831"

$ codocpy publish
Publishing Module diagram...
published at https://codoc.org/app/graph/770
```

Snippet 3: The terminal commands to publish views, and the output Alice receives.

She opens the link in her browser and observes the generated view, shown in Figure 1. The view is also accessible online at https://codoc.org/app/graph/771.

[4]https://www.scitools.com/

[5]https://www.sourcetrail.com/

[6]https://github.com/thebjorn/pydeps

[7]https://github.com/svadilfare/codoc-python/tree/v1.0.0.0

[8]https://codoc-python.readthedocs.io

[9]The source graph term is borrowed from the Symphony process [13]

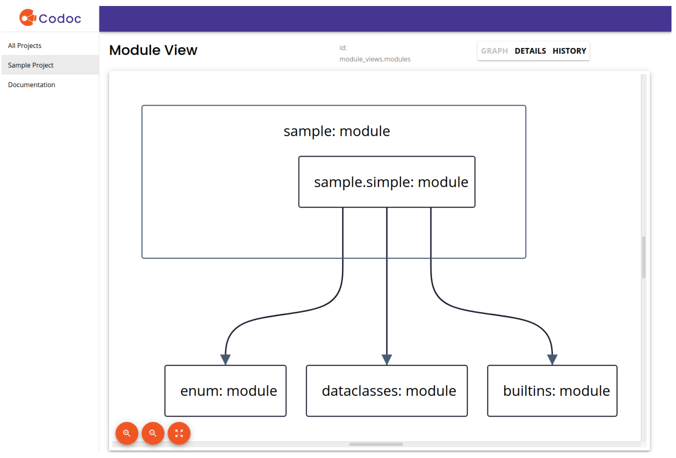

Fig. 1. The view generated by Snippet 2, as seen in the CoDoc web UI, including internal modules as well as direct dependencies

Step #6. Refining Views. Alice realizes that the view does not contain her test cases, which she also wants to visualize. After consulting the documentation, she modifies the `config.py` file into what is shown in Snippet 4. Re-

```python
from codoc import new_graph
import sample
import tests


def setup(**kwargs):
    return (
        new_graph(sample, **kwargs)
        | new_graph(tests, **kwargs)
    )
```

Snippet 4: The modified version of the configuration file, which also exports the `test` module.

running `codocpy publish` gives a new diagram, shown in figure 2 which contains the tests and dependencies of the tests module. She can also see the prior version, by inspecting the *history* tab. The updated view is accessible at https://codoc.org/app/graph/771.

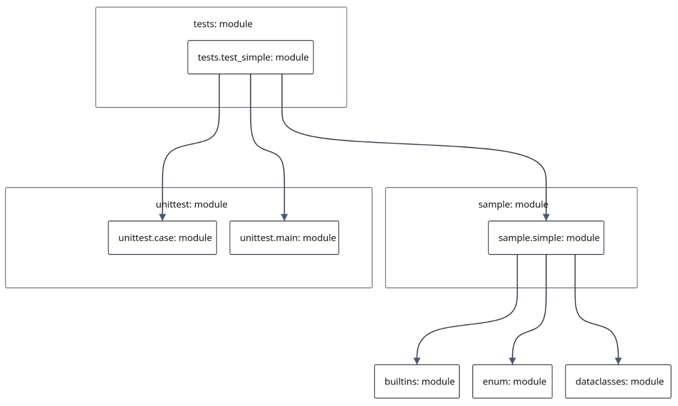

Fig. 2. The view generated by Snippet 2 with the config of Snippet 4

Step #7. Sharing Views. She then sends Bob a link to the specific view. Bob opens it, and realizes that he is missing a view that presents all the classes in the domain model of the system. He clones the project, adds a new view to the views folder, publishes it, and then notifies Alice.

IV. Evaluation Method

We ran four user tests with four different participants.

A user test is constructed by observing a given participant and their first impressions trying to setup the tool, similar to that of *Alice* in section III.

Each participant is given a link to the user documentation and asked to create a view with *Codoc*. The documentation includes a *getting-started* guide; however, the participants were not asked to follow this strictly and were encouraged to create any view that made sense for their context. We use the *Think out loud* protocol to help the participants verbalize their thoughts [8], [14], [15].

The study made it possible to observe participants' first impressions while evaluating the experience of creating architectural views and testing software practitioners' ability to articulate specific needs with the MVP.

The four tests were separated by at least a week to make it possible to minimize the likelihood that negative feedback was due to implementation details or bugs, but also meant that the system evolved between tests. The evolution of the system between tests was done from a *participatory design* perspective, where the product is polished by including potential users in the design [16]. A potential weakness is that tests are less comparable; however, we see a bigger gain as we minimize noise from issues that do not relate directly to *Codoc*, and can actually make improvements to the prototype.

The interviews were conducted over Zoom to observe the screen and participant and standardize the evaluation with participants from different countries. All interviews were recorded and evaluated afterwards.

At the start of each interview, the interviewer explained the core problem that Codoc aims to solve and explained that the meeting would be recorded. After verbal acceptance from the participant, the participant would share their screen. The interviewer provided login information to codoc.org, an API key, and a link to the relevant user documentation. The user tests was time-boxed to approximately one hour. After the test, the participant was asked to answer a series of questions that are detailed in Section V-B.

We use the answers, as well as the *think out loud* protocol, to gain insight into what are the benefits, shortcomings, and challenges when specifying architectural views via a code-driven interface.

A. Participants

All participants are males between the age range of 20-45 years old. They are briefly described below:

Participant A was found by asking for participants in a Danish forum for programmers. He is currently undertaking his 6th semester of a bachelor's in computer science at a

Danish university. He has prior work experience, both in Python and other languages. At his place of employment he answers to a project manager who is also a software practitioner. He brought for analysis an internal tool from his company, with the precondition that we would delete any artifact after the end of the test.

Participant B is a former colleague of one of the authors. He works as a freelance software engineer. He has a master's degree in *Computer Science* and has approximately five years of professional experience, both in enterprise and freelance settings. He brought for analysis a hobby CMS.

Participant C is a Canadian senior software developer and was found through an international chatroom for Python developers. He has approximately 20 years of professional experience in software development. He works in a team with 4-6 other software practitioners. He chose as subject system a collection of scripts for *advent of code*[10].

Participant D was also recruited through an international chatroom for Python developers. He is British and has the title of "Head of Engineering". He has around 20 years of experience in professional software development and works with approximately 30 other software practitioners. He has a master's degree in *Information Systems*. His analyzed project is an open-source system, to which he is a contributor.

V. Results

A. Lessons Learned Observing the Participants

Based on our observations of the participants and their thinking aloud while interacting with the tool we observe:

1) No starting view is good enough for everybody: To ease adoption, the documentation had an *introductory view* that was easy to copy-paste such that the users get up to speed quickly. This proved very useful as every participant used it. However, the challenge is that, the large variety of systems, does not guarantee that this view will be relevant for every system.

Our original introductory view initially resulted in a very large graph for Participant B that had too many nodes. The participant spent quite a long time exploring the view, while sounding overwhelmed, rather than continuing through the documentation and adding further filtering.

Based on this, we changed the introductory view code to start with stronger filters. But we ended up with a participant (C) which coincidentally had a smaller system complaining that the starting view was too simple (See Figure 3.).

Although, there probably is no introductory view to work for every system, we think it might still be preferable to err on the side of generating simpler views at first rather than overwhelming users with too complex diagrams.

2) Participants asked for more interaction mechanisms: Multiple participants ended up asking for better "navigation" mechanisms, e.g.,: Participant B said: *"I really need some way to search or something, because i really can't find the components I am looking for,"* and Participant A argued: *"It would be nice if i could select this [edge], so i could follow it through the graph"*.

[10]A yearly set of coding challenges https://adventofcode.com/

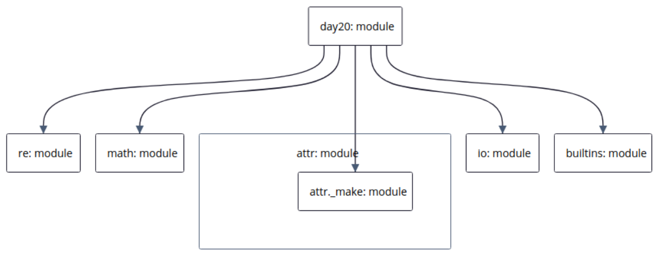

Fig. 3. The initial view generated by Participant C, showing just the module he analyzed and the external dependencies

One solution could indeed be more interactive views. However, we believe that, just as with unit testing, users need to be trained into thinking in terms of what makes an architectural view good. Then they would understand that the solution might not be more interactivity, but rather creating multiple simpler and more focused architectural views for their systems. *Codoc* supports multiple views, but our participants did not think about this when left to their own devices. One solution could be adding an advisory note to this effect when the number of graph elements exceeds some specified threshold.

3) Integration with existing documentation tool: Participant C said: *"Our existing documentation is hosted in a selfhosted Gitlab instance, and I'd want this [view] in there, without having to go through hoops"*. Publishing a URL only for the image of a view, that can be easily embedded in other documentation systems should solve this issue.

4) Incomplete Dependency Extraction Is a Deal-Breaker: Participant C observed that some of the dependencies were missing. We explained that some of the dependencies were not detected by our code analysis, in particular due to the dynamic nature of Python and the lack of type annotations his system. Participant C explained that his place of employment does not use the optional typing system, and argued that this was a requirement for any tool they would use.

5) Long Feedback Loops Are a Nuisance: In our test with Participant C, the parsing was very slow (more than three minutes), which inhibited the creation of views, as it would take a significant time to test out different filters. Even if parsing can be sped up, the limitation of having to publish a view before seeing the effect of changes to its definition is a problem that we had not thought about until testing.

Indeed, participant C requested a local visualization tool that would instantly update when modifying view functions. Such a tool would also give instant feedback when creating more complex views to determine whether it was correct and make it possible to observe views before publishing them.

6) Privacy is an issue: Participant C and Participant D were reluctant to send the views of their business systems to a third party service. For a broad adoption of such a tool a self-hosted version of the view repository would have to be considered.

B. Follow Up Questions

Here we briefly summarize the answers to the questions that we asked the participants after their interactions with *Codoc*.

1) Can you express the needed views in the existing framework? All participants responded positively regarding expressiveness and the output of their view functions. Some participants proposed additional filters that we implemented, for instance regex-based filters. Participants A and C both explicitly said that the filtering framework was a strength of the tool and that they would rather use it compared to manually graphing, while also arguing that it suited the views they needed.

2) Which tasks & activities that you do in your daily job, do you imagine that Codoc *could help with and how?* The participants were questioned about which activities would suit Codoc. The majority mentioned detecting a problematic architecture, for instance, *big ball of mud*, a disordered system with no clear architecture [17]. Participants B and D also argued for it's use to guide refactoring tasks. Participant C and Participant D suggested its usage in documentation, which was in fact our initial motivation. The fact that participants A and B did not mention documentation might reflect the amount of documentation usually written by them: they both work alone.

3) Would you use a tool like this? All participants were optimistic about the code-driven interface; however, not all participants saw a need in their current employment. Moreover, no participant saw the current solution as fulfilling, arguing for different issues, as mentioned in Section V-A. Yet, both Participant C and Participant D showed interest in the system if the existing issues were mitigated.

VI. Discussion

We discuss two aspects that are not directly raised by the participants but we think must be mentioned.

1) View Evolution: To ensure that the views stay relevant and up to date we believe that it would be good to notify the developers when changes in the code will impact a given view. To implement this a tool like *Codoc* would need a way of tracking the version for which a view is generated. Then, the CI pipeline could raise a notification about architectural views that are affected by the new commit, in a similar way in which unit testing tools raise exceptions when tests fail.

2) Visualization: The visualizations of *Codoc* are very basic. Too basic maybe (e.g. not having different glyphs for classes and modules). However, although they complained when a view was too cluttered, the participants did not complain about the visual language and conventions per se. One possible reason is that their focus on defining the views did not leave much energy for thinking critically about the visual properties of the results. Another might be some users are actually happy to leave the visualization to the tool. This aspect needs to be investigated further.

VII. Future Work

Some of the observations from the Results and Discussion sections already suggest directions of future work: supporting view evolution, a local visualizer, handling privacy. Besides these, more evaluations, and a longitudinal case study or ethnographical study where *Codoc* is used in onboarding processes or maintaining an evolving software system should be done to better understand the place of such a tool during software evolution.

Acknowledgements. We would like to thank the participants for their time and the anonymous reviewers for their feedback.

References

[1] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice," *IEEE software*, vol. 20, no. 6, pp. 35–39, 2003.

[2] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: a survey," in *Proceedings of the 2002 ACM symposium on Document engineering*, pp. 26–33, 2002.

[3] C. J. Stettina and W. Heijstek, "Necessary and neglected? an empirical study of internal documentation in agile software development teams," in *Proceedings of the 29th ACM international conference on Design of communication*, pp. 159–166, 2011.

[4] M. Leotta, F. Ricca, G. Antoniol, V. Garousi, J. Zhi, and G. Ruhe, "A pilot experiment to quantify the effect of documentation accuracy on maintenance tasks," in *2013 IEEE International Conference on Software Maintenance*, pp. 428–431, IEEE, 2013.

[5] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. Boston: Addison-Wesley Professional, 2004.

[6] L. Osterweil, "Software processes are software too," in *Engineering of Software*, pp. 323–344, Springer, 2011.

[7] B. Fitzgerald and K.-J. Stol, "Continuous software engineering and beyond: Trends and challenges," in *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, RCoSE 2014, (New York, NY, USA), p. 1–9, Association for Computing Machinery, 2014.

[8] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz, "A systematic literature review of software visualization evaluation," *Journal of Systems and Software*, vol. 144, pp. 165–180, 2018.

[9] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on software engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[10] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: Connecting software architecture to implementation," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pp. 187–197, IEEE, 2002.

[11] K. Mens, R. Wuyts, and T. D'Hondt, "Declaratively codifying software architectures using virtual software classifications," in *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No. PR00275)*, pp. 33–45, IEEE, 1999.

[12] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts, "Co-evolving code and design with intensional views: A case study," *Computer Languages, Systems & Structures*, vol. 32, no. 2-3, pp. 140–156, 2006.

[13] A. Van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva, "Symphony: View-driven software architecture reconstruction," in *Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, pp. 122–132, IEEE, 2004.

[14] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik, "How do api documentation and static typing affect api usability?," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), p. 632–642, Association for Computing Machinery, 2014.

[15] M. Piccioni, C. A. Furia, and B. Meyer, "An empirical study of api usability," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 5–14, IEEE, 2013.

[16] C. Spinuzzi, "The methodology of participatory design," *Technical communication*, vol. 52, no. 2, pp. 163–174, 2005.

[17] B. Foote and J. Yoder, "Big ball of mud," *Pattern languages of program design*, vol. 4, pp. 654–692, 1997.