Code diagrams

Types of code diagrams

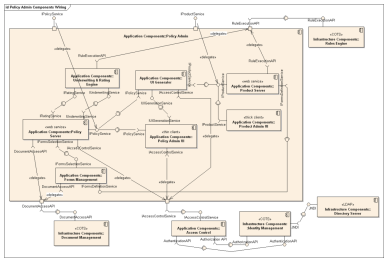
UML diagram

Ref: https://en.wikipedia.org/wiki/Unified_Modeling_Language

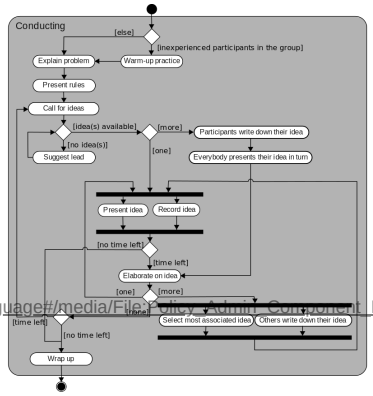
- UML = Unified Modeling Language
- a standardized way to visualize the design of a system
- very complicated, with a lot of conventions for different aspects
  - can be generated automatically

Types of UML diagrams

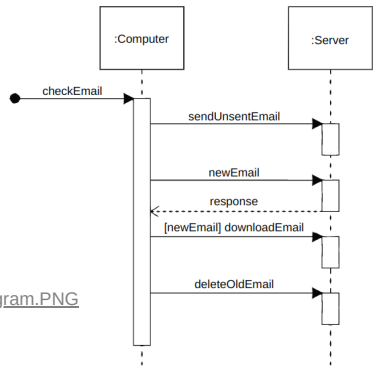
- Structure
  - static parts of a system
  - e.g. software components and their dependencies
- Behaviour
  - dynamic parts of a system
  - e.g. the activities of the system's components
- Interaction
  - flow of control and data between parts of a system
  - e.g. the sequence of messages when components communicate with each other



Structure diagram. Ref: https://en.wikipedia.org/wiki/Unified_Modeling_Language#/media/File:Activity_Conducting_Conference_Diagram.PNG



Behaviour diagram. Ref: https://en.wikipedia.org/wiki/Unified_Modeling_Language#/media/File:Activity_conducting.svg



Interaction diagram. Ref: https://en.wikipedia.org/wiki/Unified_Modeling_L

C4 diagram

Ref: https://en.wikipedia.org/wiki/C4_model

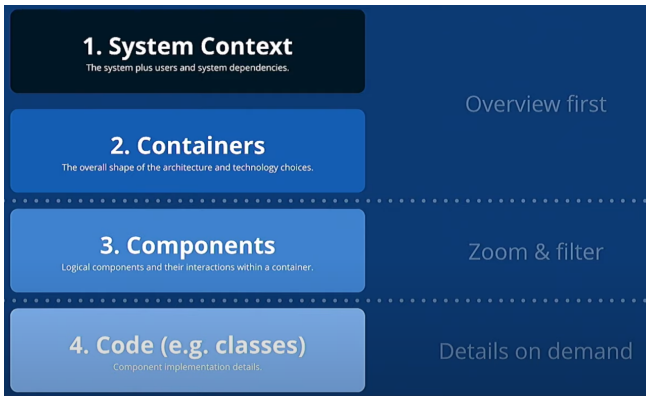
Ref: https://c4model.com

Video (35min): https://www.youtube.com/watch?v=x2-rSnhpw0g

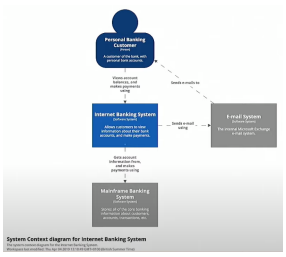
- C4 = 4 hierarchical levels of things all starting with C
- sort of a simplified version of UML that shows the bigger picture of a system
- defines some style conventions for diagrams, e.g.:
  - components should include descriptions what they do and what technology they use
  - connections between components should be worded with enough detail to give an idea of what they do and to form complete sentences
  - avoid reciprocal connections if possible
- only the most detailed level can be generated automatically

Levels of C4 diagrams

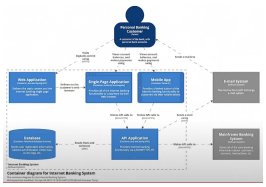
- Context
  - the general system and its relationship with other systems e.g. users
- Container
  - the applications or data stores of a system
- Component
  - the components of each application
- Code
  - basically a UML diagram



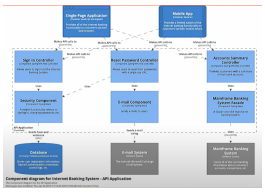
Levels of C4 diagrams. Ref: https://www.youtube.com/watch?v=x2-rSnhpw0g



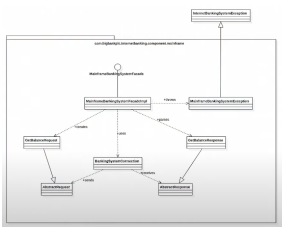
Context diagram. Ref: https://www.youtube.com/watch?v=x2-rSnhpw0g



Container diagram. Ref: https://www.youtube.com/watch?v=x2-rSnhpw0g



Component diagram. Ref: https://www.youtube.com/watch?v=x2-rSnhpw0g

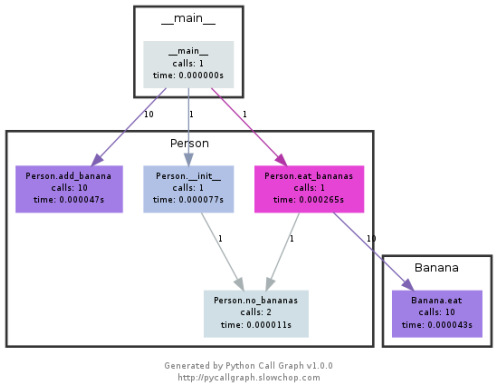


Code diagram. Ref: https://www.youtube.com/watch?v=x2-rSnhpw0g

Call graphs

Ref: https://en.wikipedia.org/wiki/Call_graph

- show how functions call each other



Callgraph for a Python program. Ref: https://en.wikipedia.org/wiki/Call_graph#/media/File:A_Call_Graph_generated_by_pycallgraph.png

How to make diagrams

Manually

diagrams.net: https://app.diagrams.net

yEd: https://www.yworks.com/products/yed

Automatically

- Python packages that do one of:
  - run your code and look at interactions
    - not great for code that requires many different user interactions
  - parse static code and infer interactions
    - might have mistakes, e.g. for dynamic programming languages, because there are extra things that can be done at runtime
      - in contrast to static programming languages, that are compiled before running
      - info on dynamic programming languages: https://en.wikipedia.org/wiki/Dynamic_programming_language
- most packages require at some point:
  - `graphviz` to produce `.dot` files
    - Homepage: https://pypi.org/project/graphviz/
    - a standard way of representing graphs
  - `pydot` to visualize `.dot` files, e.g. as `.png`
    - GitHub: https://github.com/pydot/pydot
- I only looked at packages that did static analysis

pyreverse

Homepage: https://pylint.pycqa.org/en/latest/pyreverse.html

- creates UML diagrams
- part of the `pylint` package, which tells you about various violations in code style

- Homepage: https://pylint.pycqa.org/en/latest/index.html
- has some other cool features, e.g. `symilar` finds copy-pasted code blocks in a set of files

- how to use
  - on one file:

```
# Run the following commands in a terminal

# Create dot file `classes.dot` from Python file
pyreverse filename.py
# Convert dot file to png
dot -Tpng classes.dot -o uml-diagram.png
```

    - creates `classes.dot` and `uml-diagram.png` in your current directory

  - on multiple files:

```
# Run the following commands in a terminal

# Create dot file `packages.dot` from multiple python files
pyreverse filename1.py filename2.py
# Convert dot file to png
dot -Tpng packages.dot -o uml-diagram_2.png
```

    - creates `packages.dot` and `uml-diagram_2.png` in your current directory

- pros:
  - automatically creates UML diagrams
- cons:
  - not always useful for showing relationships between parts of code
  - doesn't show you where the code comes from in your files (maybe there is an option for this?)
  - can be tricky to find relationships across multiple files if you don't have the right directory structure
    - Ref: https://github.com/PyCQA/pylint/issues/2763

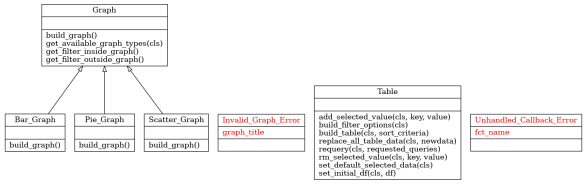


diagram from pyreverse run on one file

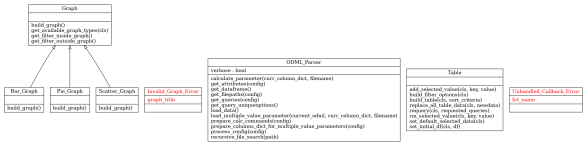


diagram from pyreverse run on multiple files

code2flow

GitHub: https://github.com/scottrogowski/code2flow

- creates call graphs
- how to use:

```
# Run the following command in a terminal

# Run on one or more files you explicitly name
code2flow filename1.py filename2.py
# or run on all files in a directory
code2flow directoryname/
```

  - creates `out.png` in your current directory
  - pros:
    - shows all the functions in your code

- has some useful colour coding
- gives line numbers of each function
- groups functions by file
- groups functions by class

- cons:
  - some colour coding is not relevant
  - doesn't show all functions
    - e.g. unused functions

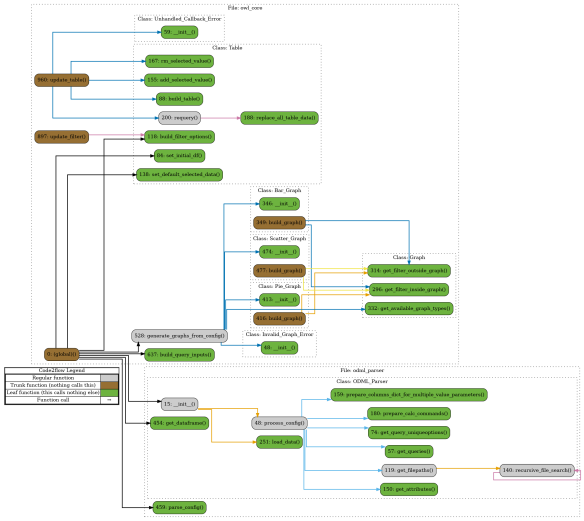


diagram from code2flow on multiple files

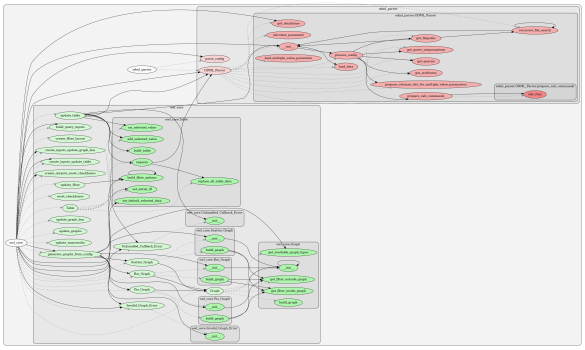
pyan and pyan3

GitHub original: https://github.com/davidfraser/pyan

GitHub development: https://github.com/Technologicat/pyan

PyPi: https://pypi.org/project/pyan3/

- `pyan3` (built from the development repo) is the most up-to-date version
- creates call graphs
- how to use:
  - there is a bug in the current version that gives errors when you run from the terminal
    - the bug is fixed in PR #65 but not included in the current released version: https://github.com/Technologicat/pyan/pull/65
    - workaround: run from a script: https://github.com/Technologicat/pyan/issues/79



```
# Include the following in a script and then run with Python

import pyan

# Don't use relative paths, they are not supported
path_to_files = 'path/to/files/*.py'
output_file_name = 'callgraph.dot'

callgraph = pyan.create_callgraph(path_to_files,
                                  #annotated=True, # optional: annotates with module and line number
                                  format='dot'
                                  )

with open(output_file_name, 'w') as f:
    f.write(callgraph)
```

```
# Run the following command in a terminal

dot -Tpng callgraph.dot -o callgraph.png
```

  - creates `callgraph.dot` and `callgraph.png` in your current directory

- pros:
  - shows all functions, even unused ones
  - function colours are intuitive
  - groups functions by file and class, and groupings are shaded to make them obvious
  - shows where functions are defined (dashed grey lines) as well as when they are called (solid black lines)
- cons:
  - difficult to show line numbers — turning on this option also shows the entire path of the file and function
  - arrows can be tricky to follow

Other packages

- pycallgraph: no longer in development (last update 7 years ago)
  - Homepage: https://pycallgraph.readthedocs.io/en/master/
  - GitHub: https://github.com/gak/pycallgraph
- python dependency graph generator: a script someone wrote (11 years old)
  - GitHub: https://gist.github.com/jbgo/1123577