Understand Architectures

Table of Contents

1. Annotations
2. Calendar
3. Directory Structure
4. Git Author
5. Git Date
6. Language
7. Visual Studio

Annotations

An architecture that organizes entities by the field values of their template annotations. Entities are grouped by template, field, and value. Entities missing a required field value are grouped under Unclassified.

Calendar

Languages: Any
Organize file entities by how recently they were modified.

Each file joins the most recent group its modification date fits: **Today**, **Yesterday**, **This Week** (the week begins on Sunday), **This Month** (since the first of the month), **This Quarter**, **This Year**, or **Earlier**. Inside each group the directory tree is reproduced, so you can also see which folders the recent changes are in.

For example, generated on a Thursday in August:

```
Calendar
  Today
    demo
      src          main.cpp
  This Week
    demo
      scripts      build.py          (modified Sunday)
  Earlier
    demo           top.py            (modified last year)
```

The dates come from each file's modification time on disk, so the groups describe your working copy: a file that a fresh checkout or a code generator touched is recent even when its content is old.

To group by commit history instead, see the Git Date architecture, which groups by

the first or last commit that touched each file, along with Git Author and Git Stability, and the metrics Days since modified and Days since creation.

Directory Structure

Languages: Any
Organize file entities by their path in the file system.

Each file is grouped by the directory that contains it, so the architecture mirrors the project's folder tree. The paths are relative to the deepest folder every project file shares, and they always keep at least one folder name - so when a file sits directly in that shared folder, the shared folder itself becomes the top of the tree.

For example, a project with these files

demo/top.py
demo/scripts/build.py
demo/src/main.cpp
demo/src/util/log.cpp
demo/src/util/log.h

builds this architecture, where each group holds the files in that one directory:

Directory Structure

demo	top.py
scripts	build.py
src	main.cpp
util	log.cpp, log.h

Every project has a Directory Structure architecture, and it is the default **Clusters** choice for cluster dependency graphs, which is why those graphs group by folder until you pick a different architecture.

Two options leave files out, which is what makes this architecture a quick way to get one that dependency rules fit. A rule like "src/api can't depend on src/ui" is natural to state against folders, but the test files under those folders reference both sides and report violations of a design that isn't actually broken. Excluding the test directories, or the test file names, gives you an architecture the rule holds on without laying one out by hand:

- **Excluded Directories** is a comma-separated list matched against the beginning of each file's directory path as the architecture shows it. Leaving out the source directory and everything below it takes demo/src in the example above, and simply src in a project whose tree starts at src.
- **Excluded File Patterns** is a comma-separated list of globbing patterns matched against file names only, such as *_test.cpp.

Excluded files are left out of the architecture instead of being grouped elsewhere, so a rule written against it never sees them. Since the architecture that every project starts with covers all of the files, an architecture you add with exclusions gets a name of its own (**Directory Structure 2** by default), and both views stay available.

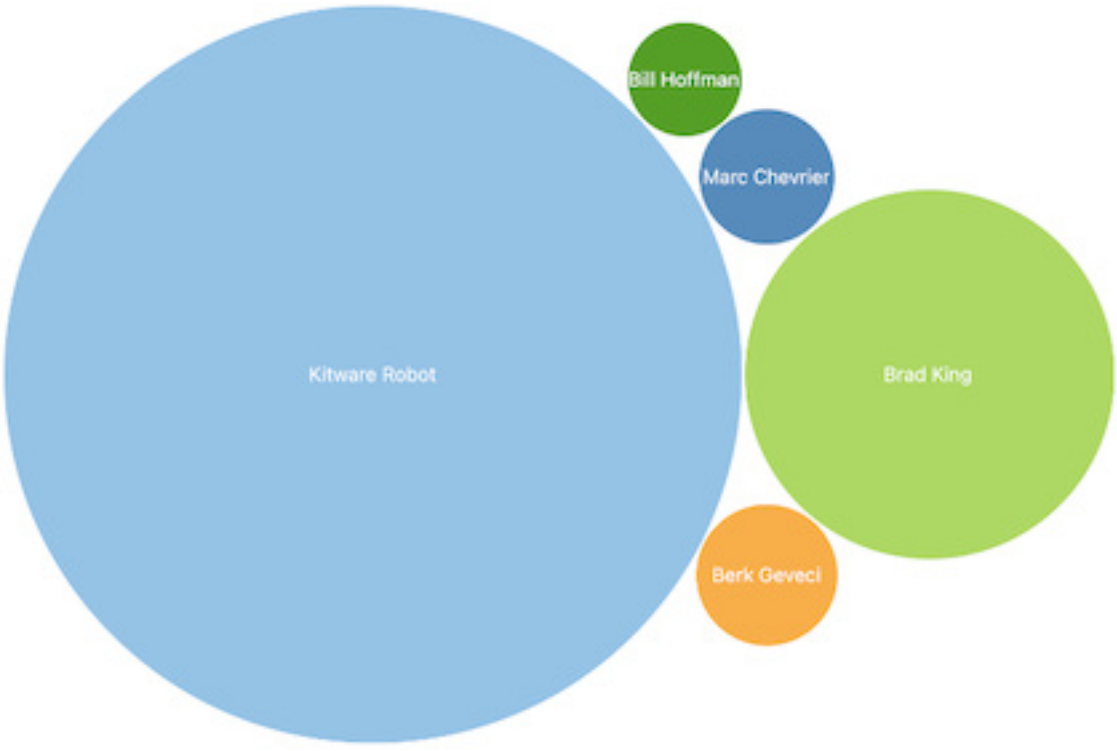
See also the Language and Calendar architectures, which group by language and by modification date and then mirror the directory tree inside each group, and the File Categorizer architecture, which sorts C/C++ files into groups like test, example, and generated code by file name pattern.

Git Author

Languages: Any
Group files by Git author.

By default, this plugin uses the most recent author, showing the last author to modify a file. It can also be configured to show the creator (first author).

Example: the "Top 5" chart of this architecture for the CMake project.

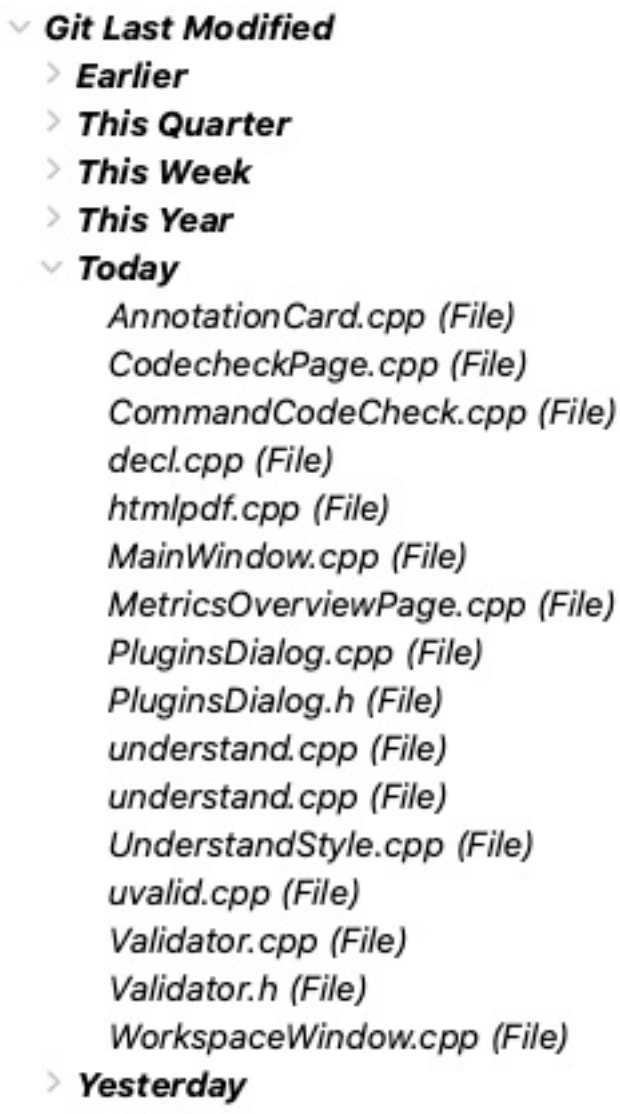


Compare to the Git Owner architecture which groups by the author with the most commits. See also the Git Solution, the blog article ↗ describing the Git plugins, the interactive report Git Authors, and the metric Authors .

Git Date

Languages: Any
Group files by Git dates.

This architecture can group by the first commit (creation date) or last commit (most recently modified date). The architecture can be structured with absolute dates (YYYY/MM) or relative dates (This Week, This Month, etc).



The full list of of relative date categories is:

- Today
- Yesterday
- This Week
- This Month
- This Quarter

- This Year
- Earlier

Compare to the Git Stability architecture which allows identifying frequently modified files, and to the built-in Calendar architecture, which uses the same relative date categories but reads each file's modification time on disk instead of its commit history. See also the Git Solution, the blog article ↗ describing the Git plugins, the interactive report Git Commits, and the metrics Commits, Days since creation and Days since modified .

Language

Languages: Any
Organize file entities by language (e.g. Ada, C, C++).

Each file is grouped by its file type, the same language Understand analyzes it as, which normally follows the file's extension and the languages the project was configured with. Inside each language the directory tree is reproduced, so you can also see which folders contribute to each language.

For example, a project with C++ and Python files:

```
Language
  C++
    demo
      src          main.cpp
        util       log.cpp, log.h
  Python
    demo           top.py
      scripts      build.py
```

This is most useful on a mixed-language project, where an architecture dependency graph shows which languages depend on which, and architecture metrics measure each language as a whole - how much of the project is C++, or how many lines of Python it carries. A single-language project produces one group holding every file.

See also the Directory Structure architecture for the folder tree on its own, and the File Namespace and CMake architectures, which group files by C++ namespace and by build target.

Visual Studio

Languages: Basic, C, C#, C++, Fortran
Organize file entities by the Visual Studio project they belong to.

When a project is created from a Visual Studio solution or project file, Understand

records the project files that build each source file. This architecture groups the files by those projects - one group per project file, named after it without the extension - and reproduces the directory tree inside each group.

For example, a solution with a Client.vcxproj and a Common.vcxproj:

```
Visual Studio
  Client
    myapp
      app          main.cpp, window.cpp
  Common
    myapp
      lib          util.cpp, util.h
```

A file that more than one project builds is grouped under each of them, which makes shared code easy to pick out: the files that show up in several groups are the ones every project pulls in.

The Visual Studio Structure (C/C++) architecture plugin is a closely related alternative. It reads a solution file you choose and reproduces the projects and filters as Solution Explorer shows them, rather than grouping by directory within each project. The CMake architecture does the equivalent for CMake targets.