Cohesion_and_coupling

September 17, 2024

0.0.1 What Is Cohesion In Software Engineering?

Cohesion represents the degree to which a part of a code base forms a logically single, atomic unit. It also can be described as the degree to which the elements of a module belong together or the number of connections inside some code unit. If the number is low, then boundaries for the unit are probably chosen badly, the code inside the unit is not logically related.

The different classes of cohesion a module may possess include:

- Functional Cohesion (grouped because they serve a single purpose)
- Logical Cohesion (elements grouped by logical category)
- Communication Cohesion (grouped because working on same data)
- Procedural Cohesion (grouped by order of execution)
- Sequential Cohesion (grouped because output of one component serves as input to the next)
- Temporal Cohesion (grouped by when in time elements execute)
- Coincidental Cohesion (grouped arbitrarily)

1. Cohesion represents the degree to which a part of a code base forms a logically single atomic unit.
2. Cohesion is an intra-module concept.
3. Cohesion depicts the module's relative functional strength.
4. High cohesion is about keeping parts of a code base that are related to each other in a single place.
5. Cohesion can be classified into the following classes: coincidental cohesion, logical cohesion, temporal cohesion, procedural cohesion, communication cohesion, sequential cohesion and functional cohesion.
6. Cohesion is a kind of natural extension of data hiding for example, the class having all members visible with a package having default visibility.
7. While designing, you need to strive for high cohesion, that is, focus on a single task with little interaction with other modules of the system.
8. Increase in cohesion is good for software.
9. High cohesion gives the best software.
10. It is possible to create fully cohesive code without introducing unnecessary coupling.

0.0.2 Cohesion in Python

Cohesion is a general sofware engineering concept. It is no different in Python than in any other language.

- Cohesion applies at the class level
  - A class should always have functional cohesion (be a data type)

- Cohesion applies at the module level
  - You may group functionality according to other cohesion models in a module

0.0.3 What Is Coupling In Software Engineering?

Coupling is the degree of interdependence between software modules. Two modules that are tightly coupled are strongly dependent on each other. On the other hand, two modules that are loosely coupled are not dependent on each other. They are henceforth referred to as uncoupled modules. Uncoupled modules have no interdependence at all within them.

There are various types of module coupling, eg.:

- No direct coupling
- Data coupling (pass data through calls)
- Stamp coupling (share common data structure)
- Control coupling (module A controls the program flow in module B)
- Common coupling (several modules have read/write on global data)
- Content coupling (a module has direct access to the internals of another module)
- External coupling (coupling through interfaces external to the program)
- Subclass coupling
- Temporal coupling
- Dynamic coupling (happens at runtime)

1. Coupling represents the degree to which a single unit is independent from others (Coupling is the number of connections between two or more units.
2. Coupling is an inter-Module concept.
3. Coupling depicts the relative independence among modules.
4. Low Coupling is about separating unrelated parts of the code base as much as possible.
5. Coupling can be classified into the following classes: data coupling control coupling stamp coupling and common coupling.
6. Making private fields, private methods and non public classes provides loose coupling.
7. While designing, you need to strive for low coupling, that is, dependence between modules should be less.
8. Increase in coupling is avoided for software.
9. Loose coupling give the best software.
10. It is impossible to achieve full decoupling without damaging cohesion.

0.0.4 Coupling in Python

Python is designed without enforced protection of internals.

- Requires discipline in coding teams
- Requires API documentation in docstrings
- Allows tests with monkey patching

0.0.5 APIs

APIs should follow best practice for loose coupling.

- The most important design criterion for an API is stability
  - Added functionality should never change the API
  - Just add new options to existing calls and add new calls, never change existing ones
  - APIs are NOT agile (within a country, you can have full freedom of movement if you have strict border controls)
- CRUD is often a good starting point for an API
  - Create
  - Retreive
  - Update
  - Delete
- However, doing this at primitive object level often incurs many roundtrips
  - You may have to pass complex data stuctures to save roundtrips
  - Use an agent model to handle packing and unpacking - this preserves the mental model of primitive objects
- A good API has functionality for introspection - machine readable
  - Version information
  - Date
  - Information about compatibility breaks
  - Call signatures
- APIs have unit and integration tests
  - Automated tests to assert that the API is working and providing correct resukts
  - Tests can serve as examples of API usage
- APIs should have no redundancy
  - Redundancy makes changes very hard
- APIs are perfect when there is nothing more to take away

0.0.6 What to put in your docstrings

- Each module and each class should have a short docstring at the top, briefly describing the purpose of the module/class.
- Each function and method that belongs to a public interface should have a docstring decribing what the function does, what inputs and output it uses.
- Generators should describe what they yield.
- Functions/methods that explicitly raise exceptions should detail which ones.

[ ]:
```
"""Single line docstring."""

"""Multiline docstrings will continue
under the first " character and have the
closing of the string on a line of its own.
It is a convention to always use triple
quotes, even if the docstring is only
one line long.
"""
```

[13]:
```
def function_with_types_in_docstring(param1, param2=None, *args, **kwargs):
    """Example function with types documented in the docstring.

    `PEP 484`_ type annotations are supported. If attribute, parameter, and
    return types are annotated according to `PEP 484`_, they do not need to be
    included in the docstring:

    Args:
        param1 (int): The first parameter.
        param2 (str): The second parameter. Optional, None if unset.
        *args: What happens to extra positional args
        *kwargs: What happens to extra keyword args

    Returns:
        bool: The return value. True for success, False otherwise.

    """
    return True
help(function_with_types_in_docstring)
```

```
Help on function function_with_types_in_docstring in module __main__:

function_with_types_in_docstring(param1, param2=None, *args, **kwargs)
    Example function with types documented in the docstring.

    `PEP 484`_ type annotations are supported. If attribute, parameter, and
    return types are annotated according to `PEP 484`_, they do not need to be
    included in the docstring:

    Args:
        param1 (int): The first parameter.
        param2 (str): The second parameter. Optional, None if unset.
        *args: What happens to extra positional args
        *kwargs: What happens to extra keyword args

    Returns:
        bool: The return value. True for success, False otherwise.
```

[14]:
```
def function_with_pep484_type_annotations(param1: int, param2: str) -> bool:
    """Example function with PEP 484 type annotations.

    Args:
        param1: The first parameter.
        param2: The second parameter.

    Returns:
        The return value. True for success, False otherwise.
```

```
    .. _PEP 484:
        https://www.python.org/dev/peps/pep-0484/
    """
    return False
help(function_with_pep484_type_annotations)
```

```
Help on function function_with_pep484_type_annotations in module __main__:

function_with_pep484_type_annotations(param1: int, param2: str) -> bool
    Example function with PEP 484 type annotations.

    Args:
        param1: The first parameter.
        param2: The second parameter.

    Returns:
        The return value. True for success, False otherwise.

    .. _PEP 484:
        https://www.python.org/dev/peps/pep-0484/
```

0.0.7 Comments

As a principle, the code should be straight forward, with clear and explicatory identifiers, making most comments redundant. However, at times, you need to explain the purpose if a piece of code - **why and what it does, not how it does it**. Then a comment is in place.

Inline comment 2 spaces before the #, one after

[15]:
```
key = 'xyzzy'  # Default key for development, in production use settings file
```

Block comment Each line starts with a # and a space. Put the comment at the same indentation level as the code, with the comment before the code.

[16]:
```
# Ensure that the SSL certificate hasn't expired before making
# the connection. This way, we reduce the number of possible
# error modes when establishing the connection.
```

0.0.8 Code formatters

There are several code formatters for Python. Some will allow you to configure lots of things, letting you have endless discussions about code formatting. There is one called **Black**, which is not configurable at all. **Use it!** Make everyone use it. Make checkins automatically use it. This will result in minimal diffs.

https://github.com/psf/black

Update There is now a formatter and linter called **ruff**. It is fully compatible with **black** but has configuration options. It is written in **rust** and is plenty faster than **black** and other linters.