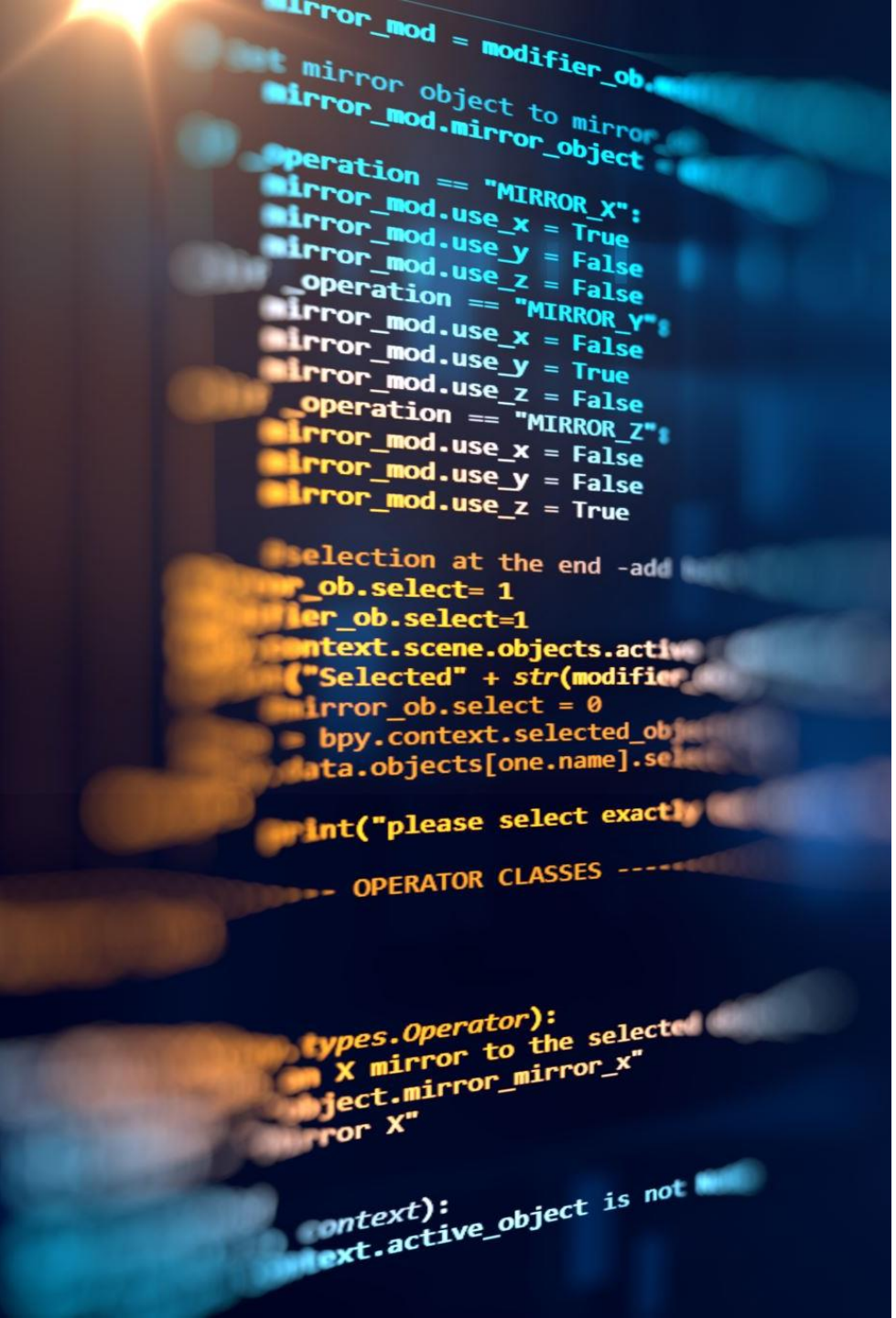


Best Practices for Intermediate Level Python Development

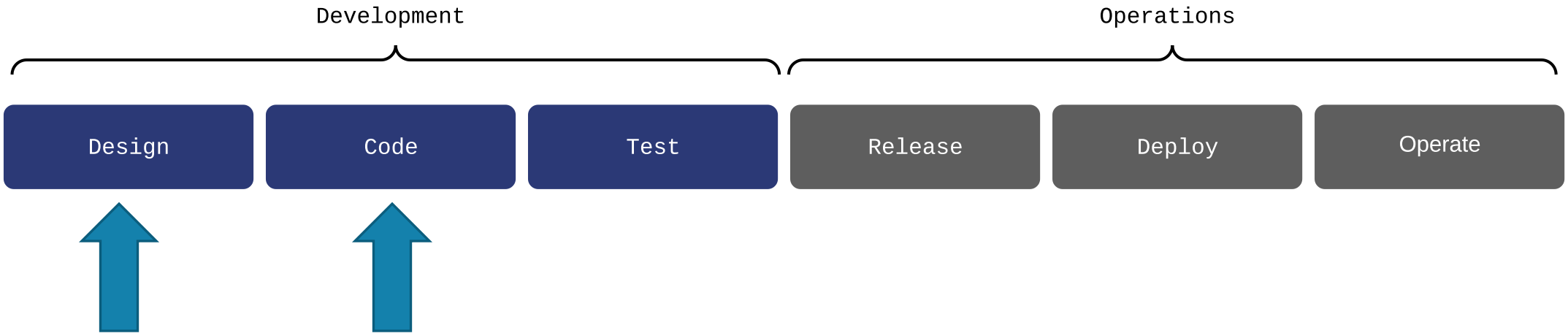
Daniel Perrefort

Center for Research Computing

University of Pittsburgh

Pitt CRC

Where Does this Fit in My Workflow?



Every stage of the software development life cycle has its own *best practices*

Today we will focus on the process of **designing** and **writing** code.

Today's Outline

1. What is a "Best Practice"?
2. Writing Clean Code With PEPs

   Break

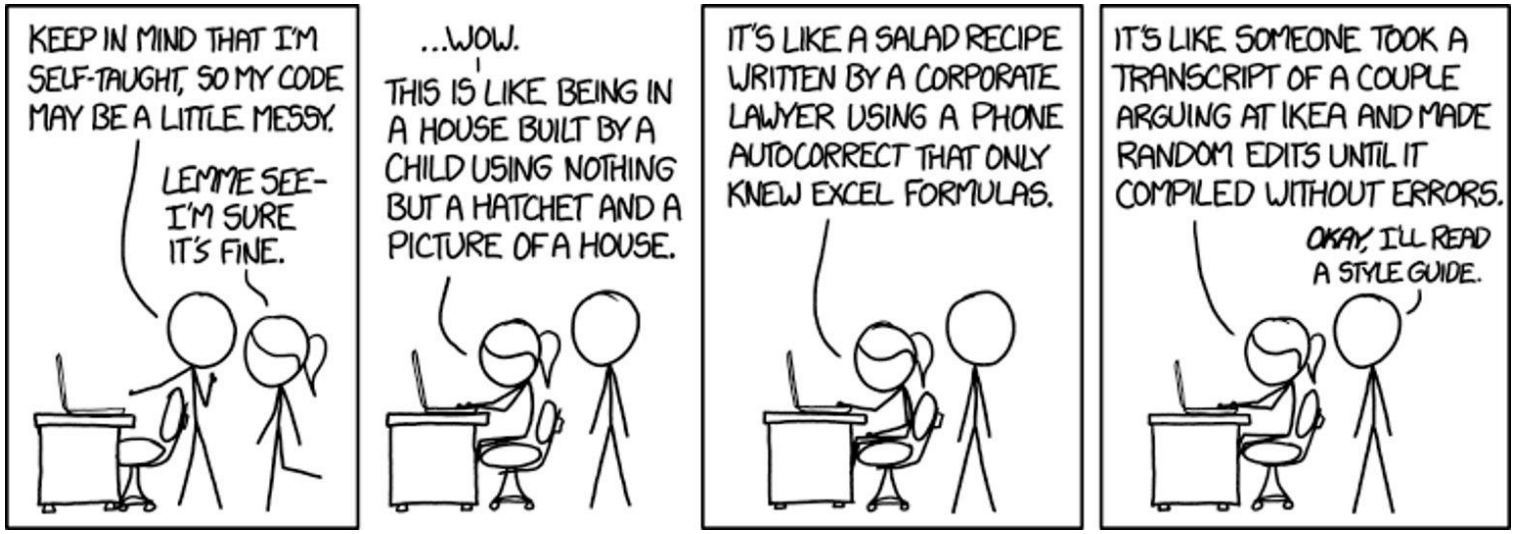
3. Common Software Design Principles

   Break

4. Tools for Easier Software Development

What Is a *Best Practice*?

What is a "Best Practice"



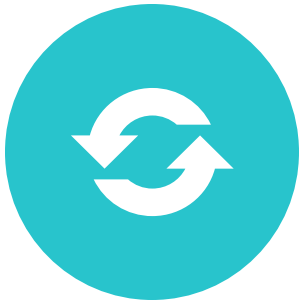
Any **procedure**, **design pattern**, or **style** that is accepted as being the most effective either by **consensus** or by **prescription**.

"Good code can be read by a professional. Great code can be read by a Student. The best code is no code at all."

—Anonymous

Tips For Following a Best Practice

Think about how you will build something before you code it



After coding, reflect on why that was a good (or not so good) approach

Work collaboratively whenever possible

Writing high quality code is an ongoing process!

Tips For **Not** Following a Best Practice

When **the guideline** makes things more difficult to understand.

When you break consistency **with surrounding code (**like legacy code).

With the **code** is no longer being maintained and you are making a small patch.

When the guideline breaks compatibility with other software.

"Best Practices" should not be followed blindly. Know when they should be ignored.

Today's Focus

- Styling Python code for readability
- Documenting your software
- Basic software design principles
- Intermediate / "Advanced" object-oriented design principles

Writing Clean Code With PEPs

Python Enhancement Protocols

"A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment." (PEP 1)

Important Fundamentals	**PEP 8:** Style Guide for Python Code
	PEP 20: The Zen of Python
	PEP 257: Docstring Conventions
Bonus PEPs	**PEP 484:** Type Hints
	PEP 498: Literal String Interpolation
	PEP 572: Assignment Expressions

PEP 20

The Zen of Python

https://peps.python.org/pep-0020/

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one - and preferably only one - obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea - let's do more of those!

```
>>> import this
```

PEP 8

Style Guide for Python Code

https://peps.python.org/pep-0008/

Topics covered by PEP8:

- Code Lay-out
- Indentation
- Maximum Line Lengths
- Using Blank Lines and Line Breaks
- File Encoding
- Imports
- Comma and Whitespace Usage
- Documentation and Comment Styling
- Naming Conventions
- Public and Internal Interface Design
- General Programming Recommendations

The big idea:

"Code is read much more often than it is written"

Why PEP 8 Matters

```
def f(n):
    if n < 0: print("Invalid"); return
    elif n == 0: return 0
    elif (
    n==1
    or n==2
    ): return 1
    return f(n-1)+f(n-2)
```

Question 1: What does this code do?

Why PEP 8 Matters

```
def fibonacci(n):
    """Returns the nth Fibonacci number"""

    if n < 0:
        print("Invalid")
        return

    elif n == 0:
        return 0

    elif n == 1 or n == 2:
        return 1

    return fibonacci(n-1) + fibonacci(n-2)
```

Question 1: What does this code do?

Question 2: How long did it take you to answer Question 1?

Things that jump out:

1. Function name + docstring provide context
2. There are 4 return cases
3. The function is recursive

The Basics...

Your probably already familiar with:

- Using 4 spaces per indentation level (not tabs!)
- Putting two blank lines before functions and classes
- Limiting line lengths to:
  - 79 characters for code
  - It is okay to increase the line length limit (Be consistent)

PEP 8 – Using Booleans

- Booleans are already booleans – they don't need comparisons
- For sequences, (e.g., a lists), use the fact empty sequences are false

```
my_boolean = True

# Incorrect
if my_boolean == True:
    do_something()

# Incorrect
if my_boolean is True:
    do_something()

# Still Incorrect
if len(my_list) != 0:
    do_something()
```

```
my_boolean = True

# Correct for sequences and booleans
if my_boolean:
    do_something()

# An empty list is False
if my_list:
    do_something()
```

PEP 8 – Using **is**

- Use `is` when comparing singletons
- Use `is not` instead of `not ... is`
- Remember `None` is a singleton

```
# Incorrect
if foo == None:
    do_something()


# Also Incorrect
if not foo is None:
    do_something()
```

```
# Correct
if foo is None:
    do_something()

# Correct
if foo is not None:
    do_something()
```

PEP 8 – Using **with**

- Also known as a "context manager"
- Use with to handle opening/closing files, database transactions, etc.

```
# Incorrect
for i in range(10):
    input_file = open(f"file_{i}.txt")
    input_file.readline()
    input_file.close()
```

```
# Better
for ind in range(10):
    with open(f"file_{ind}.txt") as input_file:
        input_file.readline()

# Even Better
directory = Path(".")
for file in directory.glob("file_*.txt"):
    with file.open() as input_file:
        ...
```

PEP 8 – Using **try/except**

- Know "Look before you leap" (LBYL) vs. "Easier to Ask Forgiveness than Permission" (EAFP)
- Use explicit exception catching (avoid bare exceptions)
- Keep `try` statements as simple as possible

```
# Incorrect
try:
    import platform_specific_module
    my_function()

except:
    platform_specific_module = None
```

```
# Correct
try:
    import platform_specific_module

except ImportError:
    platform_specific_module = None

else:
    my_function()
```

PEP 8 – Using **lambda**

- Avoid using anonymous functions
- Common exceptions:
  - Short, single use functions
  - Wrapping types as callables
  - Functions defined in a narrow scope

```
# Incorrect
double = lambda x: 2 * x
```

```
# Correct
def double(x):
    return 2 * x
```

PEP 8 – Variable Naming Conventions

TYPE	NAMING CONVENTION	EXAMPLES
Function	Use lowercase words separated by underscores.	function, my_function
Variable	Use lowercase letters or word, or words separated with underscores. (I.e., snake_case)	x, var, my_variable
Class	Start each word with a capital letter. Do not separate words with underscores. (I.e., CamalCase)	Model, MyClass
Method	Use lowercase words separated with underscores.	class_method, method
Constant	Use an uppercase single letter, word, or words separated by underscores.	CONSTANT, MY_CONSTANT
Module	Use short lowercase words separated with underscores.	module.py, my_module.py
Package	Use short lowercase words without underscores.	package, mypackage

PEP 8 – Variable Naming Example

```
GLOBAL_VAR = 1


def my_method():
    print(GLOBAL_VAR)


class MyClass:

    def __init__(self, my_var=2):
        self.my_var
        self._private_var

    def my_method(self):
        ...
```

PEP 8 – Whitespace

Functions and methods are styled mostly the same way. →

Notice the single space before methods – not double space. →

```
GLOBAL_VAR = 1


def my_method():
    print(GLOBAL_VAR)


class MyClass:

    def __init__(self, my_var=2):
        self.my_var
        self._private_var

    def my_method(self):
        ...
```

← Space around equals

← No space around equals

PEP 257

Docstring Conventions

https://peps.python.org/pep-0257/

The aim of this PEP is to standardize the high-level structure of docstrings: what they should contain, and how to say it (without touching on any markup syntax within docstrings). The PEP contains conventions, not laws or syntax.

"A universal convention supplies all of maintainability, clarity, consistency, and a foundation for good programming habits too. What it doesn't do is insist that you follow it against your will. That's Python!"

—Tim Peters on comp.lang.python, 2001-06-16

If you violate these conventions, the worst you'll get is some dirty looks. But some software (~~such as the~~ Docutils ~~docstring processing system~~ ~~PEP 256~~, ~~PEP 258~~) will be aware of the conventions, so following them will get you the best results.

What is a Docstring

```
def fibonacci(n):
    """Returns the nth Fibonacci Number"""

    if n < 0:
        print("Invalid")

    elif n == 0:
        return 0

    elif n == 1 or n == 2:
        return 1

    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

- String literal as the first statement in a module, function, class, or method
  - Assigned to the __doc__ attribute
- Describe **what** a function/class does not **how** it works
  - Exception: Uncommon technical details
- Always use """triple double quotes""" for docstrings
  - Use r""" if you use backslashes in your docstrings
  - Use u""" for Unicode docstrings
- Use a blank line after docstring
- Docstrings can be *single-line* or *multi-line*

Single-Line Function Docs

- Include a single line docstring at minimum
- Use for really obvious cases.
- They should really fit on "one line"

```
# Wrong: Don't document how
def average(a, b):
    """Add a + b and then divide by 2"""

# Wrong: Don't document signatures
def average(a, b):
    """function(a,b) -> list"""
```

```
def average(a, b):
    """Return the average of a and b"""
```

Multi-Line Function Docs

- Start with a one-line description and add **as necessary**:
  - A longer explanation
  - Arguments/Returns
  - Raised exceptions

Note how the documentation describes the behavior - not the implementation.

```
def connect_to_next_port(self, minimum):
    """Connects to the next available port.

    Connections are left opened until closed manually

    Args:
      minimum (int): A port value greater or equal to 1024

    Returns:
      The new port value

    Raises:
      ConnectionError: If no available port is found.
    """
```

Writing Class Docs

```
class Square:
    """A class used to represent a geometric Square

    Attributes:
        length (float): Side length of the square

    Methods:
        area (int): Return the area of the square
    """

    def __init__(self, length):
        """Create a square with the given side length

        Args:
            length (float): Side length of the square
        """
```

- Class docstring summarize class behavior
  - List the **public** methods/attributes
  - Required subclass interfaces (if abstract)
- __init__ (or __new__) documents construction
  - Don't document private methods/attributes
- Subclasses should summarize interfaces differences
  - Use “override” for overwritten methods
  - Use “extend” to indicate a call to super

Writing Class Docs In Reality

```
class Square:
    """A class used to represent a geometric Square"""

    def __init__(self, length):
        """Create a new square with a given side length

        Args:
            length (float): Side length of the square
        """

    def _private_helper(self, length):
        # This doesn't have to be publicly documented,
        # but docs are still useful for other developers
```

- Avoid duplicate documentation
- Document class, constructor, and all public methods
- Implement "full docs" in code developed for a user base

Writing File Level Docs

- For standalone scripts, include
  - Include usage and command line syntax
  - Include functionality and environment variables.
  - Can be elaborate (several screens full)
  - Must be sufficient for a new user to use the command
  - Should be quick reference for the sophisticated user.
- For modules:
  - Describe module purpose
  - Include submodules / subpackages
  - Include classes, exceptions and functions
  - Limit summaries to one-line each.
- Follow the same style as other docstring

Writing **Useful** Comments

- Code can be its own documentation.
- Commenting out code blocks is confusing
- Avoid the "royal we"

```
# Open the file
with file.open() as input_file:
    …

# We iterate over array elements
for element in array:
    # print(element)
    # element += 1
    # element = element.copy()
    ...
```

```
# Load directory contents into database
with file.open() as input_file:
    …
```

Break

Common Software Design Principles

Design Principles Overview

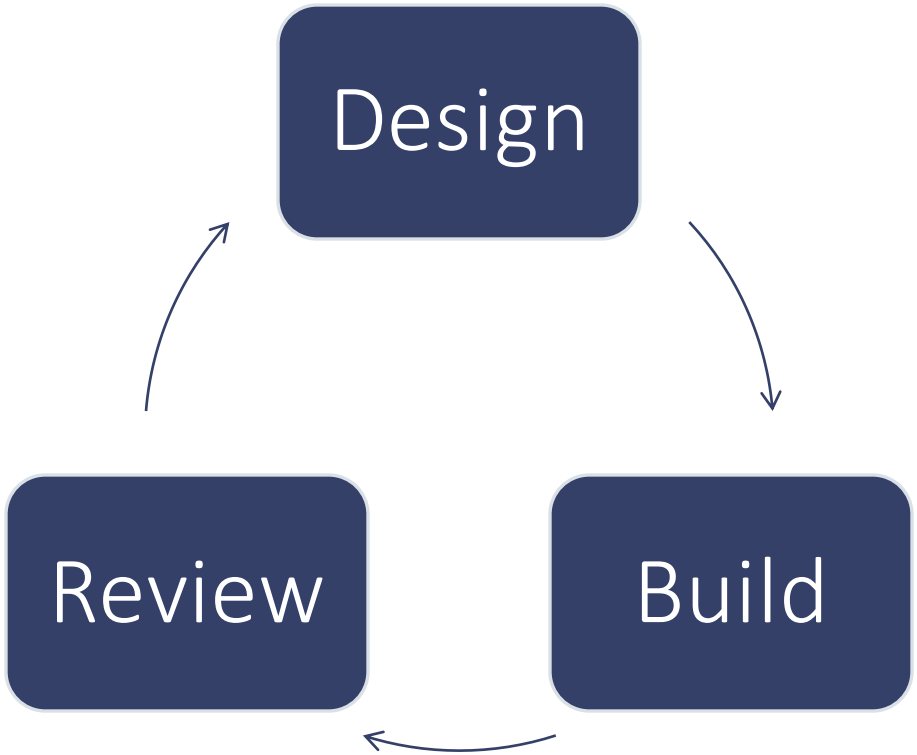
FUNDAMENTALS

- Big Design Up Front (BDUF)
- Keep It Simple (KISS)
- Principle of Least Surprise
- You Aren’t Going To Need It (YAGNI)
- Don't Repeat Yourself (DRY)

OBJECT-ORIENTED DESIGN (OOD)

- **S** - Single-responsibility Principle
- **O** - Open-closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle

Big Design Up Front



- When designing code:
  - Design the architecture first
  - Divide requirements into stages based on priority
  - Repeat BDUF principle at each stage
- Bigger projects = bigger designs
- Design however works for you
  - Draw it out on a whiteboard
  - Lay out your design in UML
  - Draft some exploratory code

Keep It Simple (KISS)

- What is "simple" code?
  - Simple code is usually easy
  - Simple code is straightforward

- Related Concepts:
  - Coupling: How much do modules depend on each other?
  - Cohesion: How well the modules belong together. Simple: Composed of few, well defined parts with low coupling and high cohesion

- Simple code has only as many parts as necessary with **low coupling** and **high cohesion**

Keep It Simple (KISS)

- Keep your methods short
- Focus on crucial/critical methods before adding frills
- Methods should only address one problem at a time
- Break up the code into smaller blocks as you go
- Avoid excessive **branching**, deep **nesting**, or **complex class structures**

Principle of Least Surprise

- Code usage should be intuitive and obvious
- Some of this is naming practices:

```
def square(a):
```

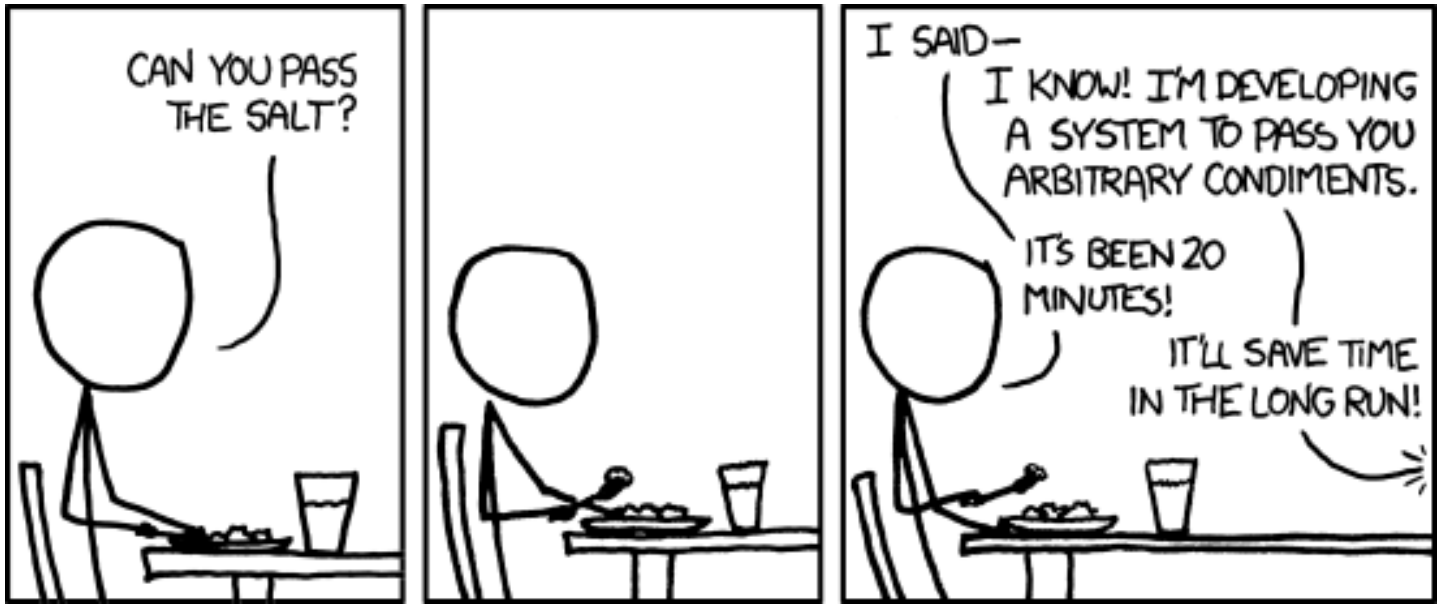
```
def square_area(side_length):
```

- Some of it is implementation:

```
def subtract(x, y):
    """Subtract two numbers"""

    return y - x
```

```
def subtract(x, y):
    """Subtract two numbers"""

    return x - y
```

You Aren’t Going To Need It (YAGNI)



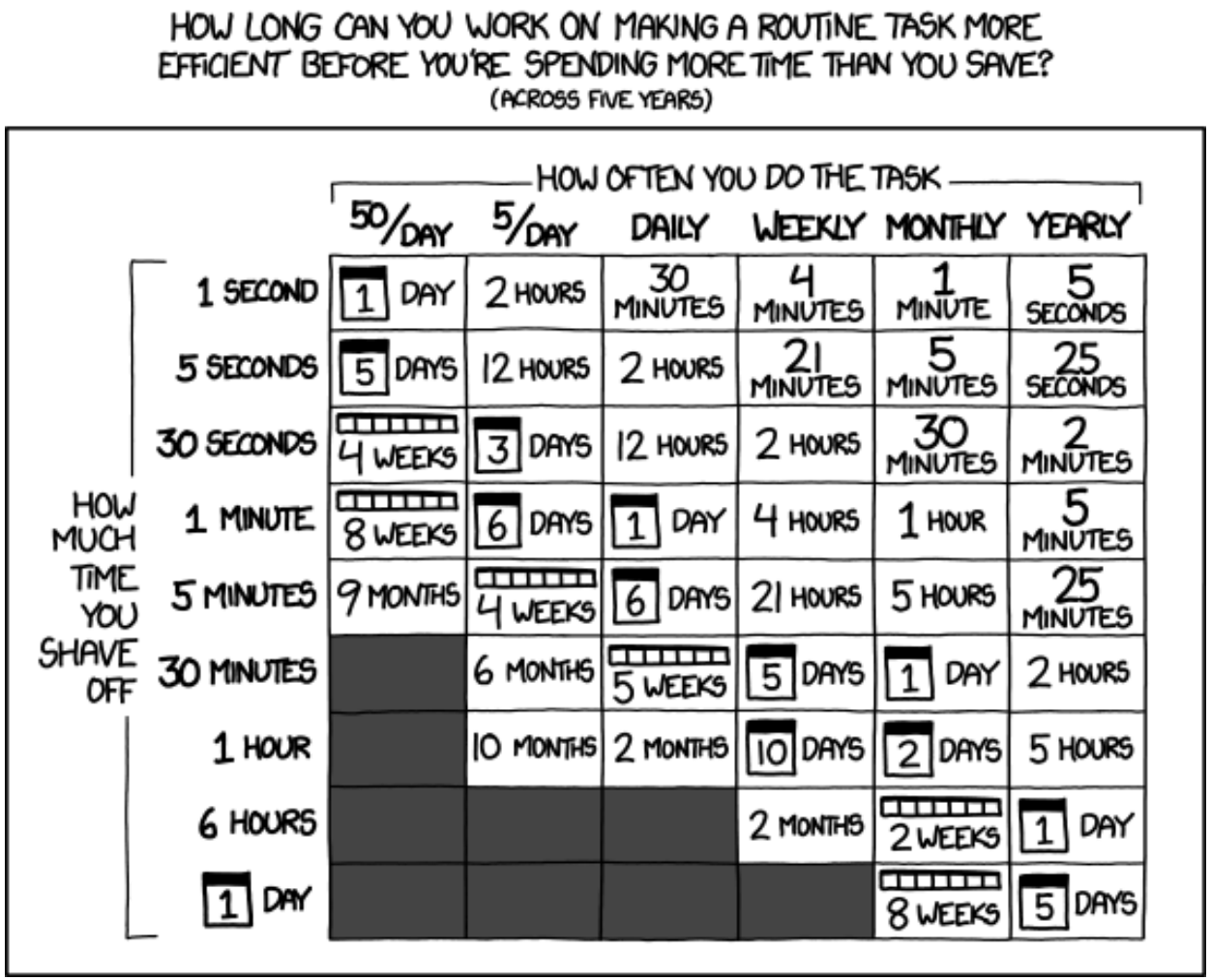
You Aren’t Going To Need It (YAGNI)

That new feature probably **wont**

- Save any time in the long run
- Justify the added complexity
- Cover real world edge cases

But it probably **will**

- Eat up your time
- Add overhead (testing / maintaining)
- Break and cause a headache



Don't Repeat Yourself (DRY)

- Duplicate code should be moved into a dedicated function/method
- Duplicate code is WET (write everything twice)

- Example scenario with WET code:
  1. You implement a new feature
  2. The code for that feature gets copy and pasted repeatedly
  3. You find a bug in the feature
  4. You go on a bug hunt to find every instance of reused code
  5. You hope you found every instance of the problem

- Example scenario with DRY code:
  1. You implement a new feature
  2. You find a bug in the feature
  3. You fix the bug

Fundamental Principles (Review)

FUNDAMENTALS

- Big Design Up Front (BDUF)
- Keep It Simple (KISS)
- Principle of Least Surprise
- You Aren’t Going To Need It (YAGNI)
- Don't Repeat Yourself (DRY)

SOLID Design Principles

- **S** - Single-responsibility Principle
- **O** - Open-closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle

Single Responsibility Principle (SRP)

- Every module, class, or function should be responsible for a single functionality, and it should encapsulate that part.

- In simpler terms:
  - SRP applies at all levels of code (functions, classes, modules, packages)
  - Each "unit of code" should be responsible for a single task
  - Each unit should be properly encapsulated

- SRP **does not** argue for giant-monolithic structures. It's the opposite!

"A class should have only one reason to change"

-Robert C. Martin

SRP Example


```
def download_data(url):
    """Download project data"""
```

```
def average_yield(data):
    """Return average stock yield"""
```

```
def upload(processed_data, DB):
    """Load data into project DB"""
```

SRP Example


```
class Extract:

    def __init__(self):
        self._data = None

    def authenticate(self, user_key):
        """Log in to remote server"""

    def download_data(self, url):
        """Download project data"""
```

```
class Transform:

    def average_yield(data):
        """Return value metrics"""

    ... # Other calculations
```

```
class Load:

    def upload(data, DB):
        """Load data into DB"""
```

Question: Should the `authenticate` step be in its own class? Why?

Open/Closed

- Objects should be open for extension but closed for modification
  - A class should be extendable without modifying the class itself

- Open/Closed benefits from:
  - Clean inheritance structures (assuming SRP)
  - Polymorphism in dependency classes
  - Low coupling between classes

Open/Closed Example

```
class Square:
    """Stores geometric properties for a square"""

    def __init__(self, length):
        self.length = length


class Circle:
    """Stores geometric properties for a circle"""

    def __init__(self, radius):
        self.radius = radius
```

```
class AreaCalculator:

    def total_area(self, shape_arr):
        """Return the total area for a collection of shapes"""

        total_area = 0
        for shape in shape_arr:
            if isinstance(shape, Square):
                total_area += shape.length ** 2

            elif isinstance(shape, Circle):
                total_area += pi * shape.radius ** 2

        return total_area
```

Open/Closed Example

```
class Square:
    """Stores geometric properties for a square"""

    def __init__(self, length):
        self.length = length

    def area(self):
        return self.length ** 2

class Circle:

    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return pi * self.radius ** 2
```

```
class AreaCalculator:

    def total_area(self, shape_arr):
        """Return the total area for a collection of shapes"""

        return sum(shape.area() for shape in shape_arr)
```

Notice how this solution also follows the SRP.

Liskov Substitution

Parent classes should be replicable with their child classes

Note:

We don't actually expect random code substitutions. This is more of a "guiding principle" for designing good inheritance structures.

In practicality:

- Avoid child classes that have little in common with the parent class
- Aim for high **cohesion**

Liskov Substitution Example

You have been tasked with writing two classes - one representing a `Square` and one representing a `Rectangle`.

Define these classes in a way that:

1. One class inherits from another
2. Each class has a method for the `area` of the shape
3. The classes obey Liskov Substitution

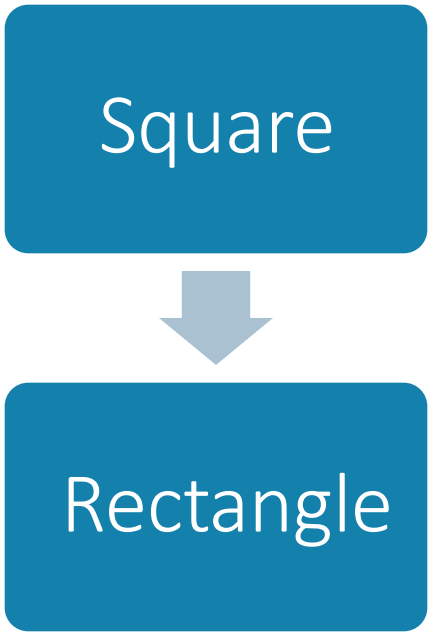
Liskov Substitution Example

You have been tasked with writing two classes - one representing a `Square` and one representing a `Rectangle`.

Define these classes in a way that:

1. One class inherits from another
2. Each class has a method for the `area` of the shape
3. The classes obey Liskov Substitution

Option 1

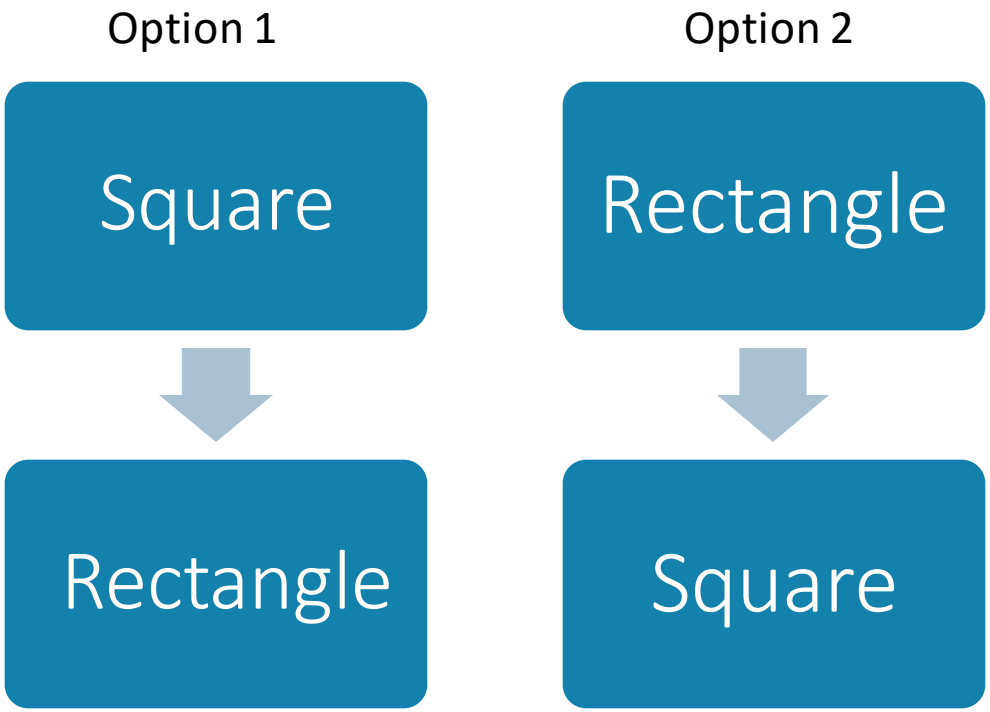


Liskov Substitution Example

You have been tasked with writing two classes - one representing a `Square` and one representing a `Rectangle`.

Define these classes in a way that:

1. One class inherits from another
2. Each class has a method for the `area` of the shape
3. The classes obey Liskov Substitution



Liskov Substitution Example

You have been tasked with writing two classes - one representing a `Square` and one representing a `Rectangle`.

Define these classes in a way that:

1. One class inherits from another
2. Each class has a method for the `area` of the shape
3. The classes obey Liskov Substitution

```
class Rectangle:

    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width
```

Liskov Substitution Example

You have been tasked with writing two classes - one representing a `Square` and one representing a `Rectangle`.

Define these classes in a way that:

1. One class inherits from another
2. Each class has a method for the `area` of the shape
3. The classes obey Liskov Substitution

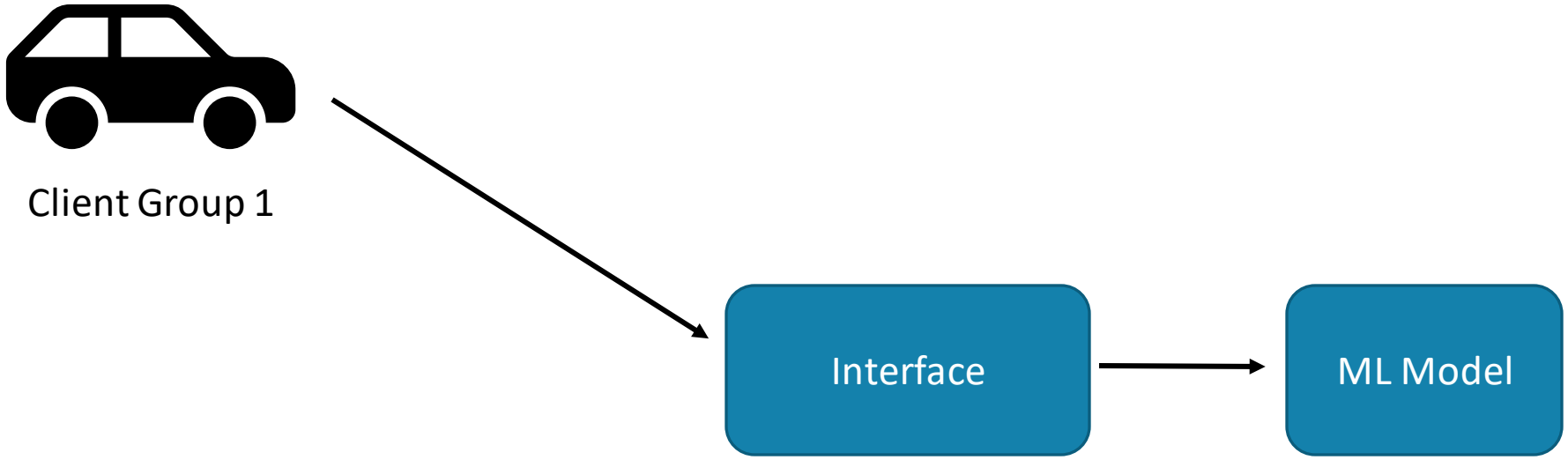
```
class Rectangle:

    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width


class Square(Rectangle):

    def __init__(self, length):
        super().__init__(length, length)
```

Interface Segregation

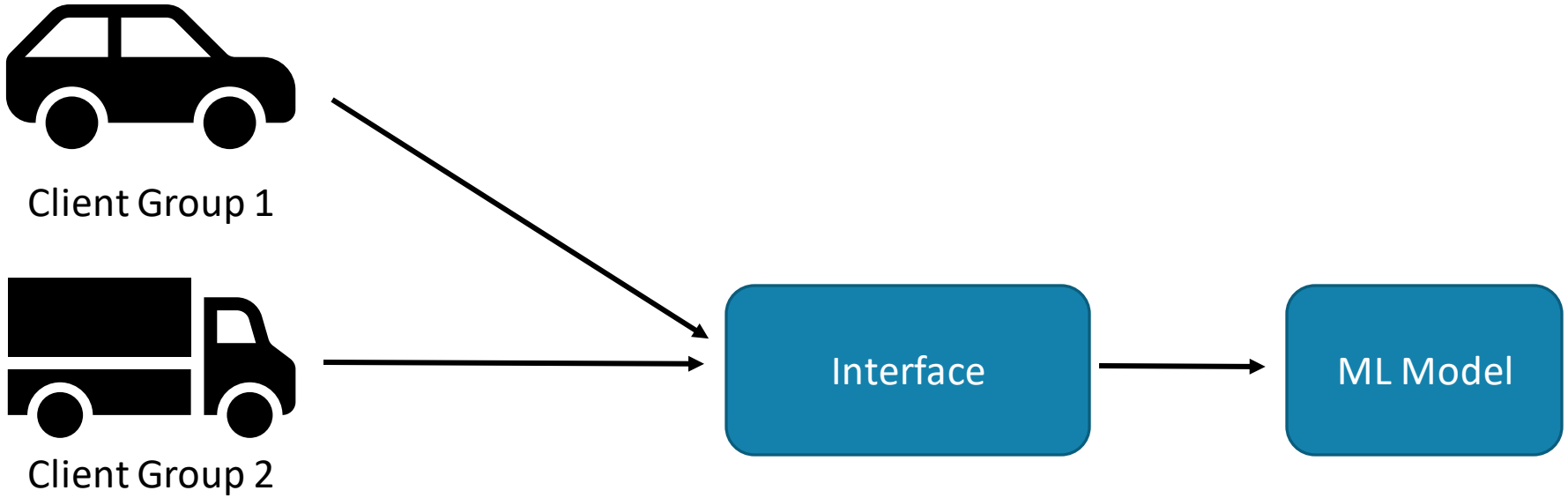
- An interface is a set of abstractions:
  - `Square.area()`
  - `Square.perimiter()`
  - `Square.width()`

- Clients should not be required to use interfaces they don't need
  - Most applicable to large projects
  - Avoid giant, monolithic interfaces
  - Rely on smaller, client specific interfaces

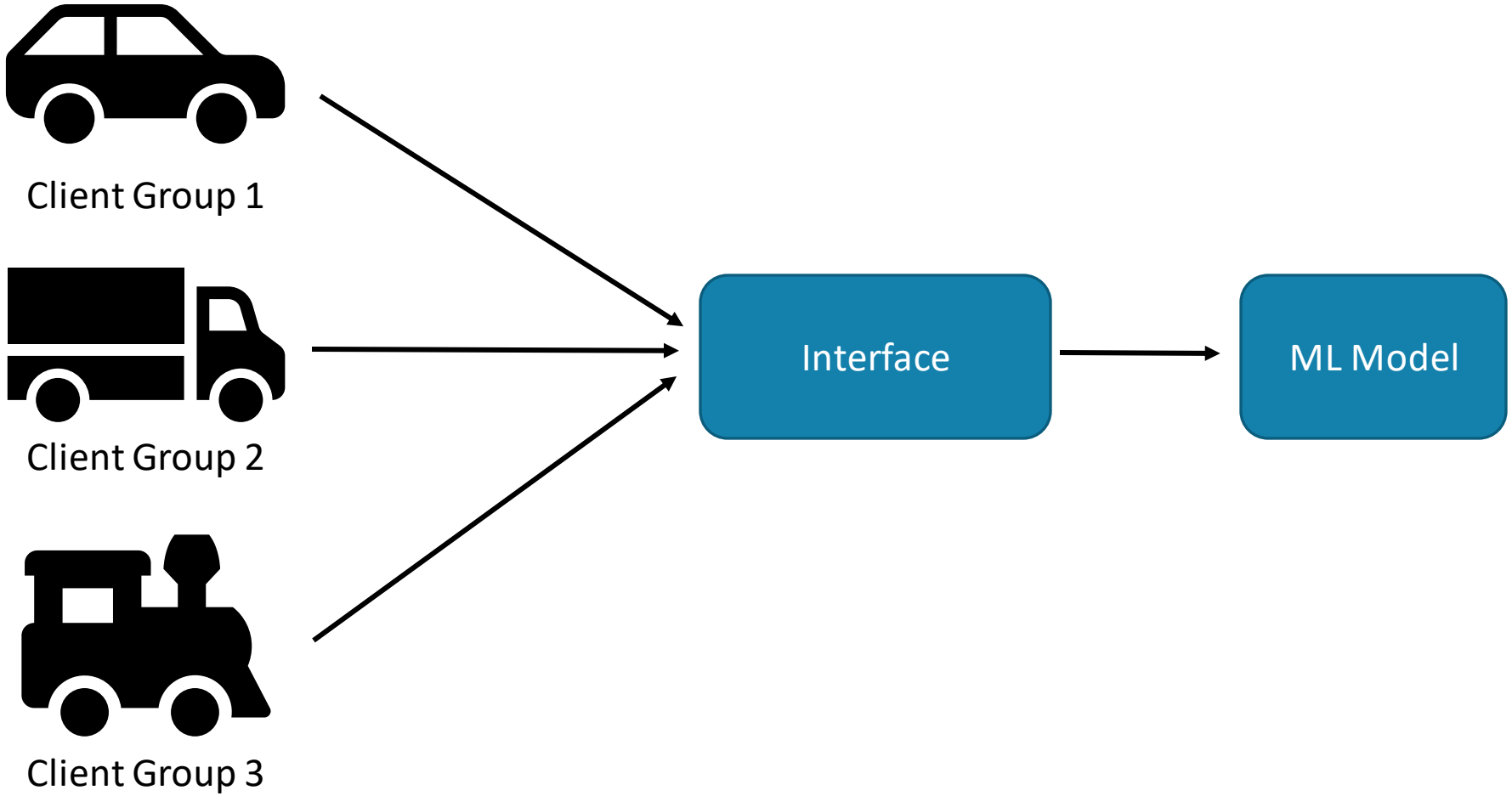
Interface Segregation Example



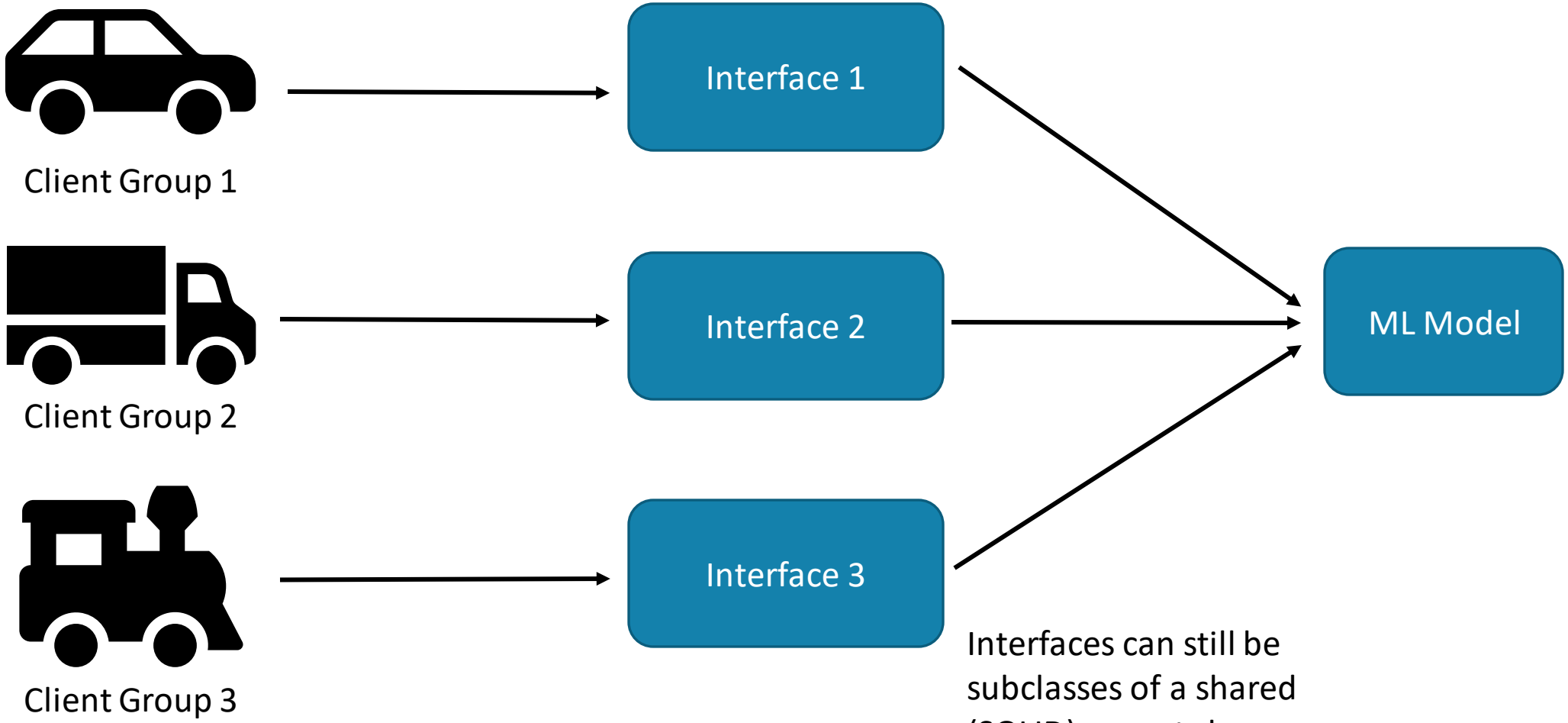
Interface Segregation Example



Interface Segregation Example



Interface Segregation Example



Dependency Inversion Principle

- High-level constructs should not rely on low level implementations
  - Both should depend on abstractions (e.g., interfaces).
- Abstractions should not depend on details.
  - Details (implementations) should depend on abstractions.
- In simple terms: Rely on abstractions

Dependency Inversion Example

```
class Square:
    """Stores geometric properties for a square"""

    def __init__(self, length):
        self.length = length


class Circle:
    """Stores geometric properties for a circle"""

    def __init__(self, radius):
        self.radius = radius
```

```
class AreaCalculator:

    def total_area(self, shape_arr):
        """Return the total area for a collection of shapes"""

        total_area = 0
        for shape in shape_arr:
            if isinstance(shape, Square):
                total_area += shape.length ** 2

            elif isinstance(shape, Circle):
                total_area += pi * shape.radius ** 2

        return total_area
```

Dependency Inversion Example

```
class Square:

    def __init__(self, length):
        self.length = length

    def area(self):
        return self.length ** 2

class Circle:

    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return pi * self.radius ** 2
```

```
class AreaCalculator:

    def total_area(self, shape_arr):
        """Return the total area for a collection of shapes"""

        return sum(shape.area() for shape in shape_arr)
```

Notice how this solution also follows the SRP and Open/Closed.

Solid Principles Review

OBJECT-ORIENTED DESIGN (OOD)

- **S** - Single-responsibility Principle
- **O** - Open-closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle

Break

Tools for Easier Software Development

Enforcing PEP 8

- Command line tools for PEP 8 are also available
  - Pylint: http://pylint.pycqa.org/
  - Flake8: https://flake8.pycqa.org/
- PEP8 inspection is built into many Integrated Development Environments (IDEs)
- Jupyter Plugins:
  - Python Black: https://github.com/drillan/jupyter-black

Using Pylint

```
def Fibonacci(n):
    """Returns the nth Fibonacci Number"""
    if n < 0:
        print("Invalid")

    elif n == 0:
        return 0

    elif n==1 or x == 2:
        return 1

    else:
        return Fibonacci(n-1) + Fibonacci(n-2)
```

```
$ pylint example.py
```

Using Pylint

```
def Fibonacci(n):
    """Returns the nth Fibonacci Number"""
    if n < 0:
        print("Invalid")

    elif n == 0:
        return 0

    elif n==1 or x == 2:
        return 1

    else:
        return Fibonacci(n-1) + Fibonacci(n-2)
```

```
$ pylint example.py
************* Module example
example.py:1:0: C0114: Missing module docstring (missing-module-docstring)

example.py:1:0: C0103: Function name "Fibonacci" doesn't conform
                       to snake_case naming style (invalid-name)

example.py:1:14: C0103: Argument name "n" doesn't conform to
                        snake_case naming style (invalid-name)

example.py:3:4: R1705: Unnecessary "elif" after "return" (no-else-return)

example.py:10:17: E0602: Undefined variable 'x' (undefined-variable)

example.py:1:0: R1710: Either all return statements in a function should return
                       an expression, or none of them should.
                       (inconsistent-return-statements)

--------------------------------
Your code has been rated at -1.11/10
```

What is an IDE?

An Integrated Development Environment (IDE) is a software application designed to **maximize** a programmer's **productivity** by providing a **comprehensive set of tools** and facilities.

- Wikipedia

Are Jupyter Notebooks an IDE?

Yes... kind of ...

- Autocomplete
- Syntax highlighting
- Code execution
- Cross language support (HTML, Markdown)
- Plugin support

But no, not really ...

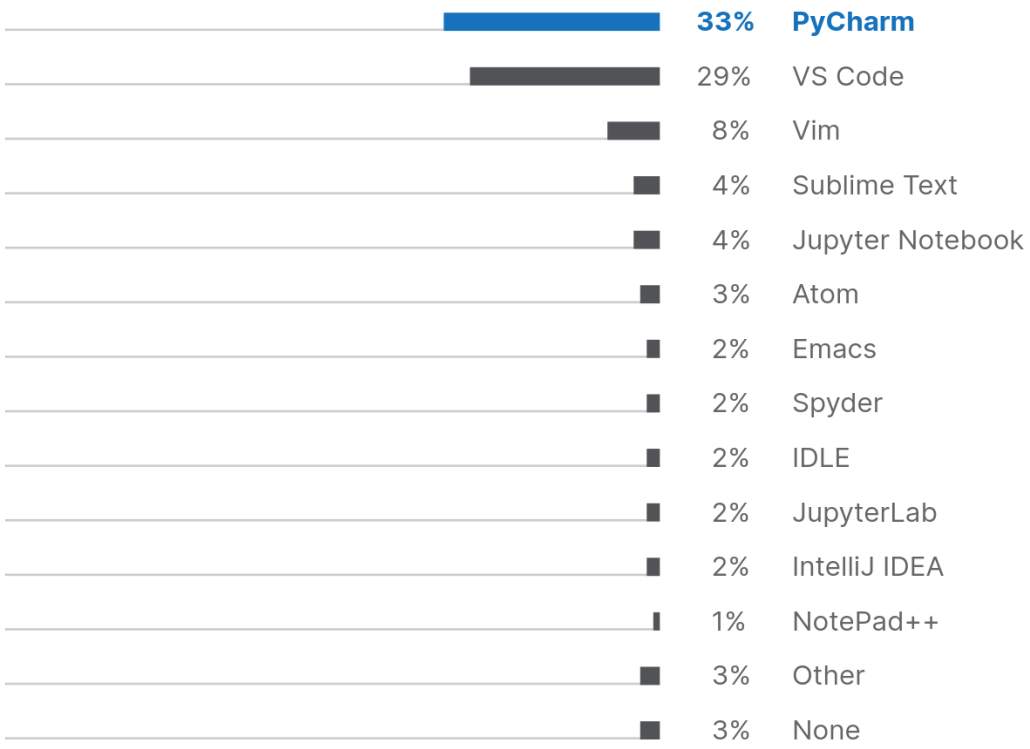
- No cross file support
- No integrated test suite / profiling tools
- No major refactoring or code search tools
- Missing dozens of other useful features

Common IDE Features

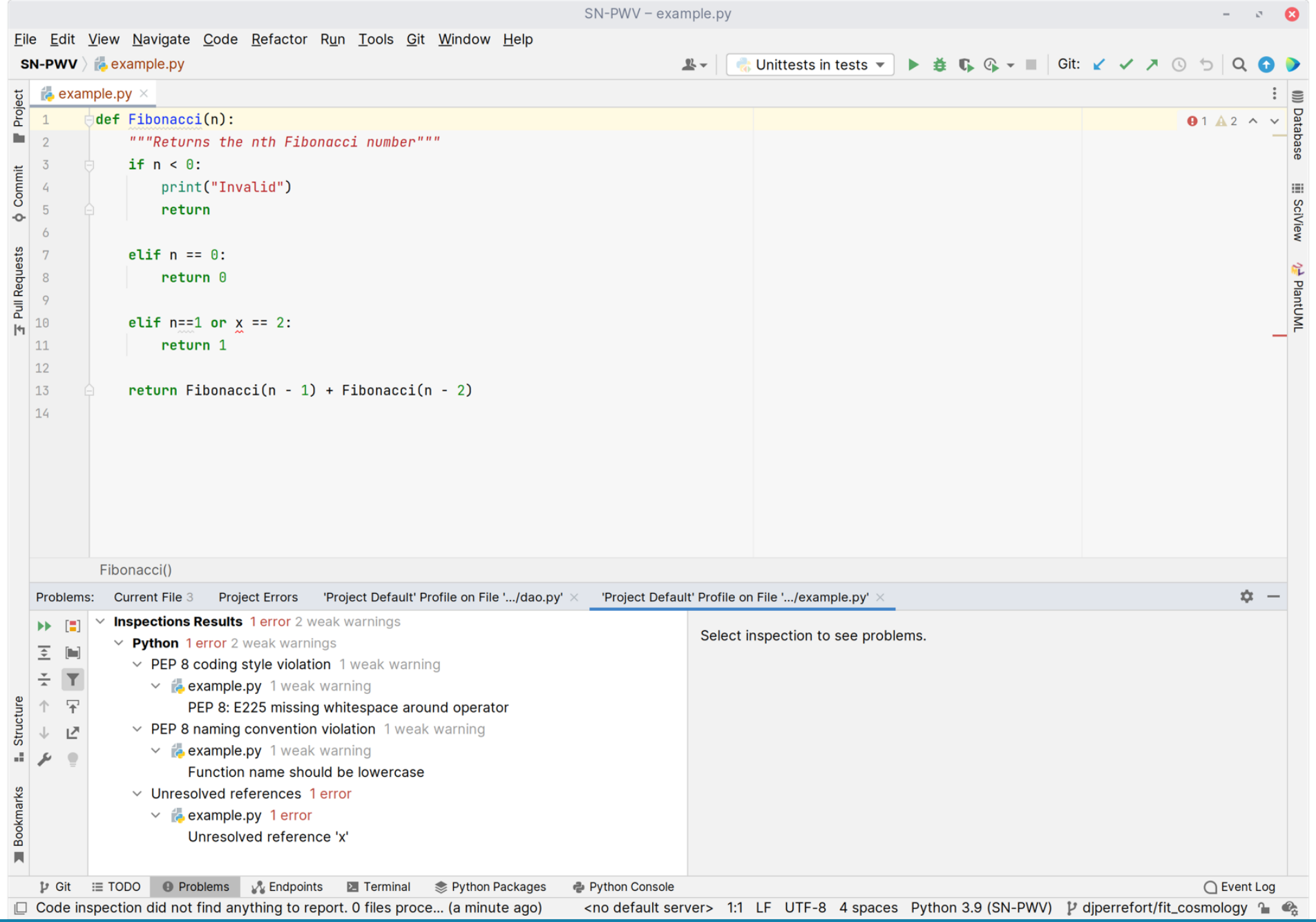
- Refactoring
- Real time syntax and argument checking
- Automatic code formatting
- Automatic docstring templates
- Code navigation
- GitHub Integration
- Test suite integration
- Test coverage reports
- Profiling

- All of your tools in one place (Terminal, File Explorer, Code Editor, GitHub UI, ...)
- Optimization Suggestions
- Built-in debugging tools
- Auto code generation (getters and setters)
- File navigation
- Command line interface
- PEP 8, 257, and 484 integration

Picking an IDE



- IDEs are generally language specific
  - Some support for "secondary" languages
- > 75% of developers write code in an IDE
  - Jetbrains 2020 developer survey


SN-PWV – example.py
File Edit View Navigate Code Refactor Run Tools Git Window Help
SN-PWV example.py
Unittests in tests
Git:
example.py
1 def Fibonacci(n):
2 """Returns the nth Fibonacci number"""
3 if n < 0:
4 print("Invalid")
5 return
6
7 elif n == 0:
8 return 0
9
10 elif n==1 or x == 2:
11 return 1
12
13 return Fibonacci(n - 1) + Fibonacci(n - 2)
14
Fibonacci()
Problems: Current File 3 Project Errors 'Project Default' Profile on File '.../dao.py' 'Project Default' Profile on File '.../example.py'
Inspections Results 1 error 2 weak warnings
Python 1 error 2 weak warnings
PEP 8 coding style violation 1 weak warning
example.py 1 weak warning
PEP 8: E225 missing whitespace around operator
PEP 8 naming convention violation 1 weak warning
example.py 1 weak warning
Function name should be lowercase
Unresolved references 1 error
example.py 1 error
Unresolved reference 'x'
Select inspection to see problems.
Git TODO Problems Endpoints Terminal Python Packages Python Console Event Log
Code inspection did not find anything to report. 0 files proce... (a minute ago)
<no default server> 1:1 LF UTF-8 4 spaces Python 3.9 (SN-PWV) djperrefort/fit_cosmology
Project Commit Pull Requests Structure Bookmarks Database SciView PlantUML

Enforcing Coding Principles

- Develop Software Collaboratively
  - Get feedback from senior developers
  - Hold eachother to established guidelines

- Software inspection Tools
  - Great in a CI setting, but take a lot of upfront configuration
  - www.codacy.com
  - www.codeclimate.com

GitHub.com

- A cloud-based VCS hosting system with integrated utilities for building and deploying software

- GitHub is built on git and provides web-based wrappers for git features

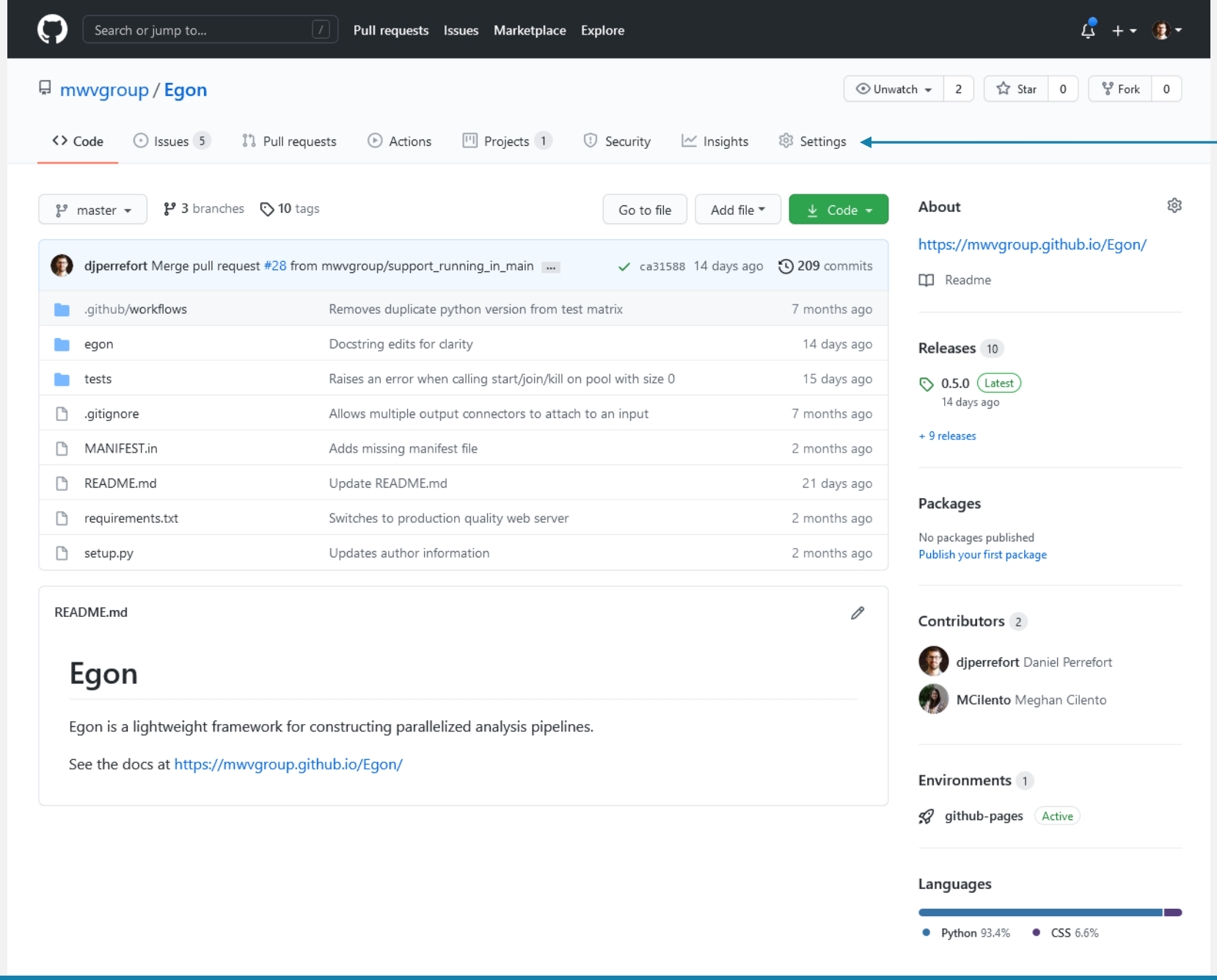
- Some great **GitHub** features
  - Graphical interface for visualizing source code, commit history, branches, etc.
  - Collaborative platform for reviewing and approving source code changes
  - Robust permissions management settings
  - Support for automated tasks
  - Easier conflict resolution than git (usually)

Current branch

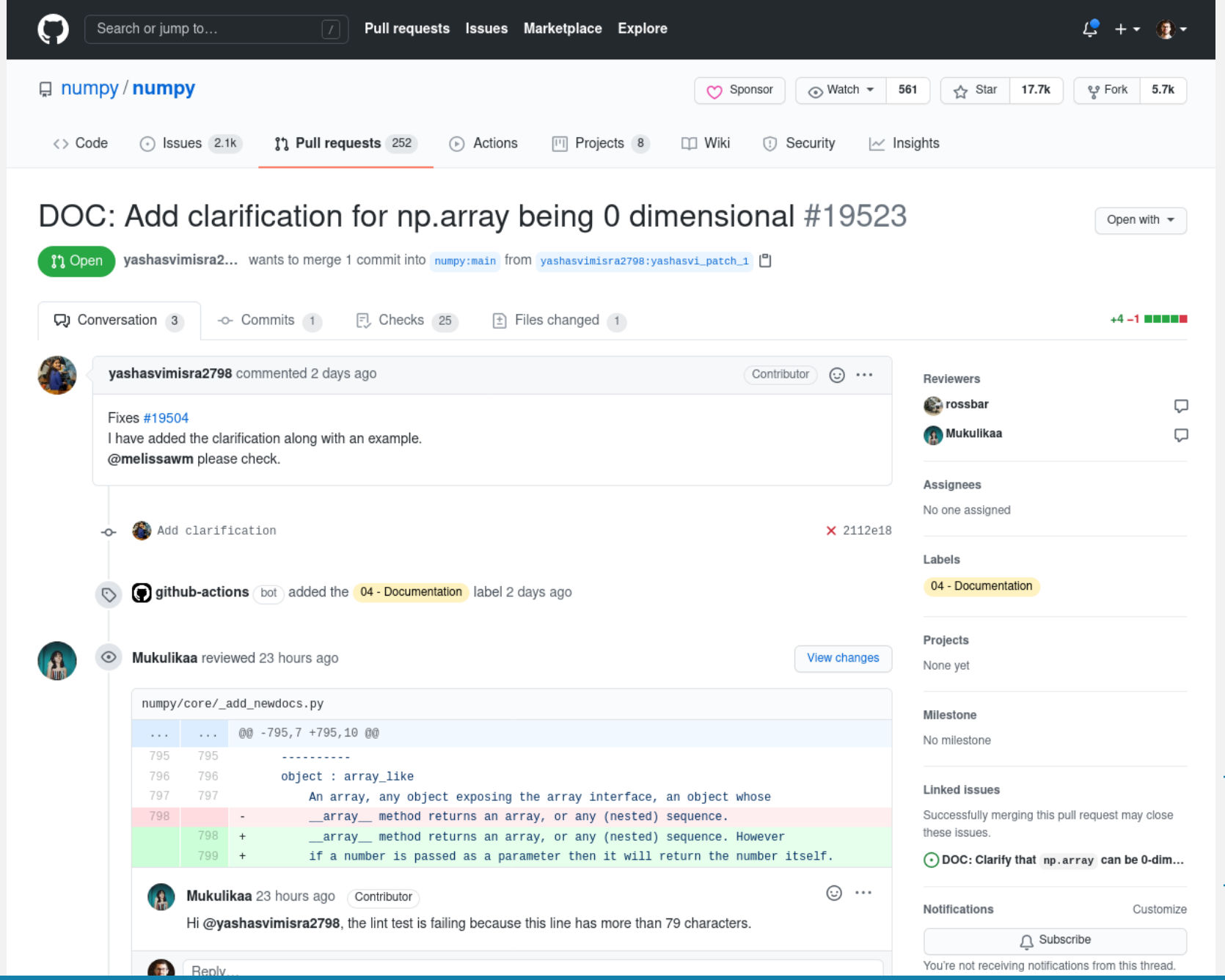
Last Commit

Files included in the repository

Contents of the README File



Other repository data



PR title

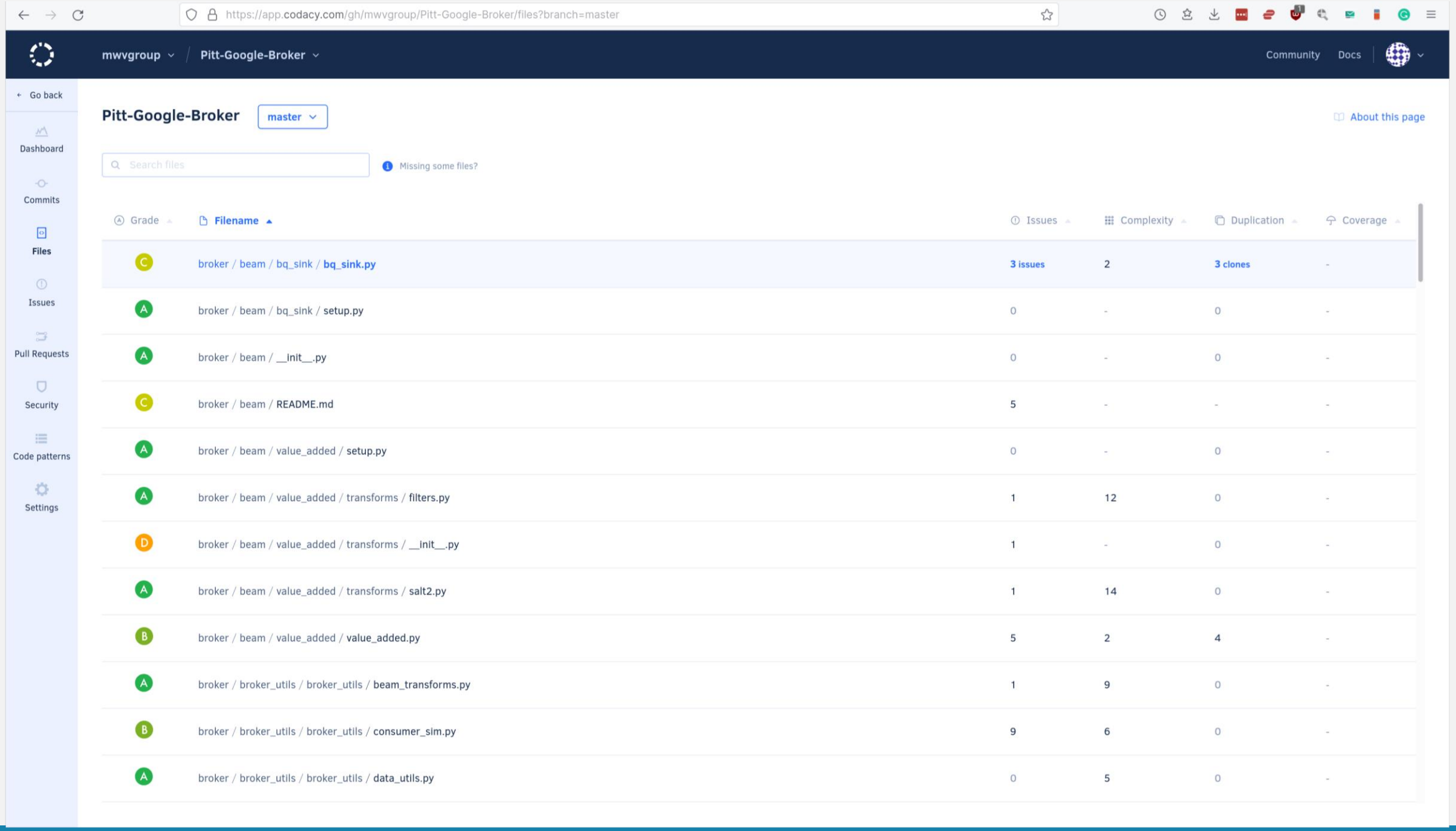
PR status

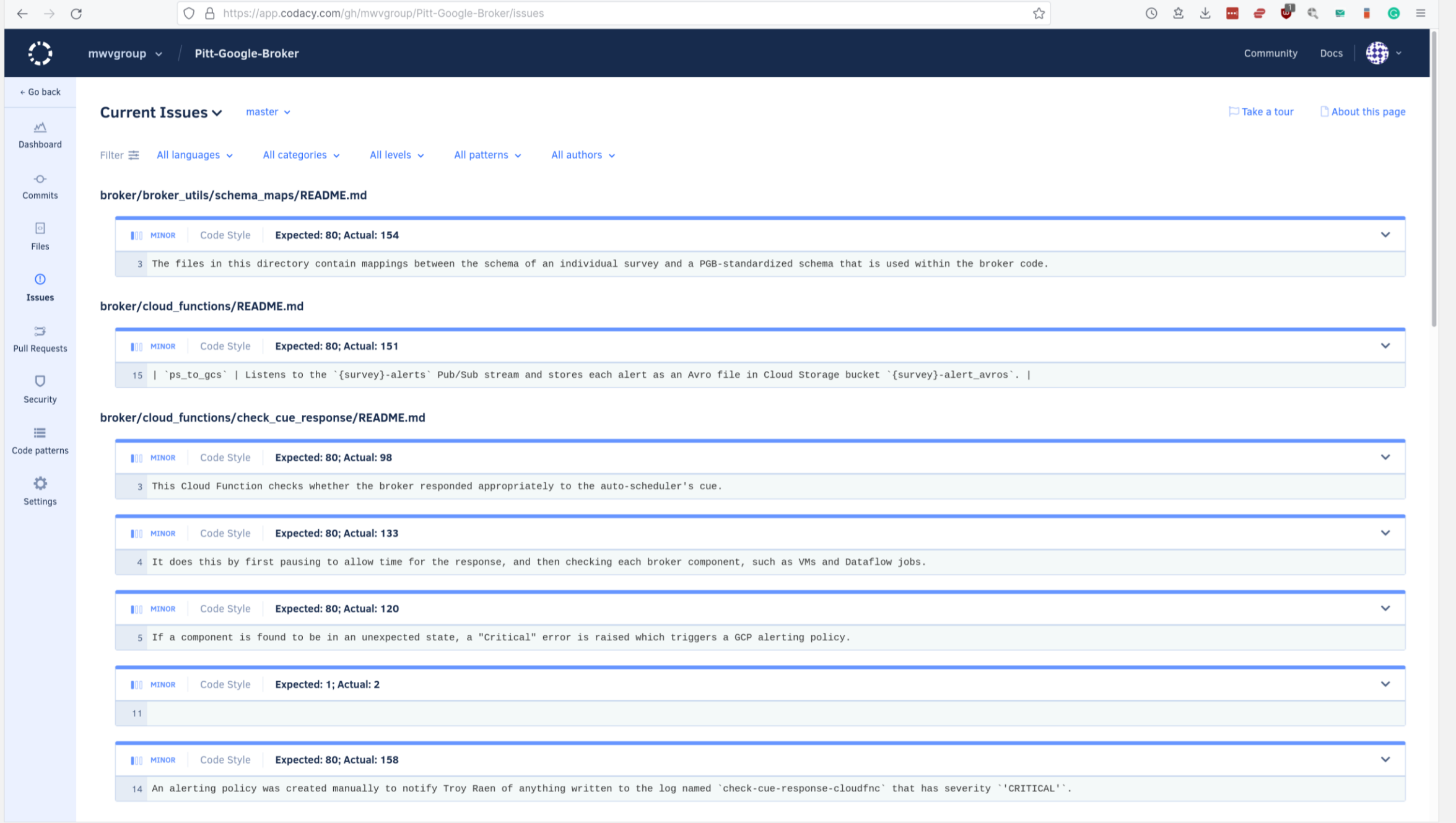
Description of proposed changes

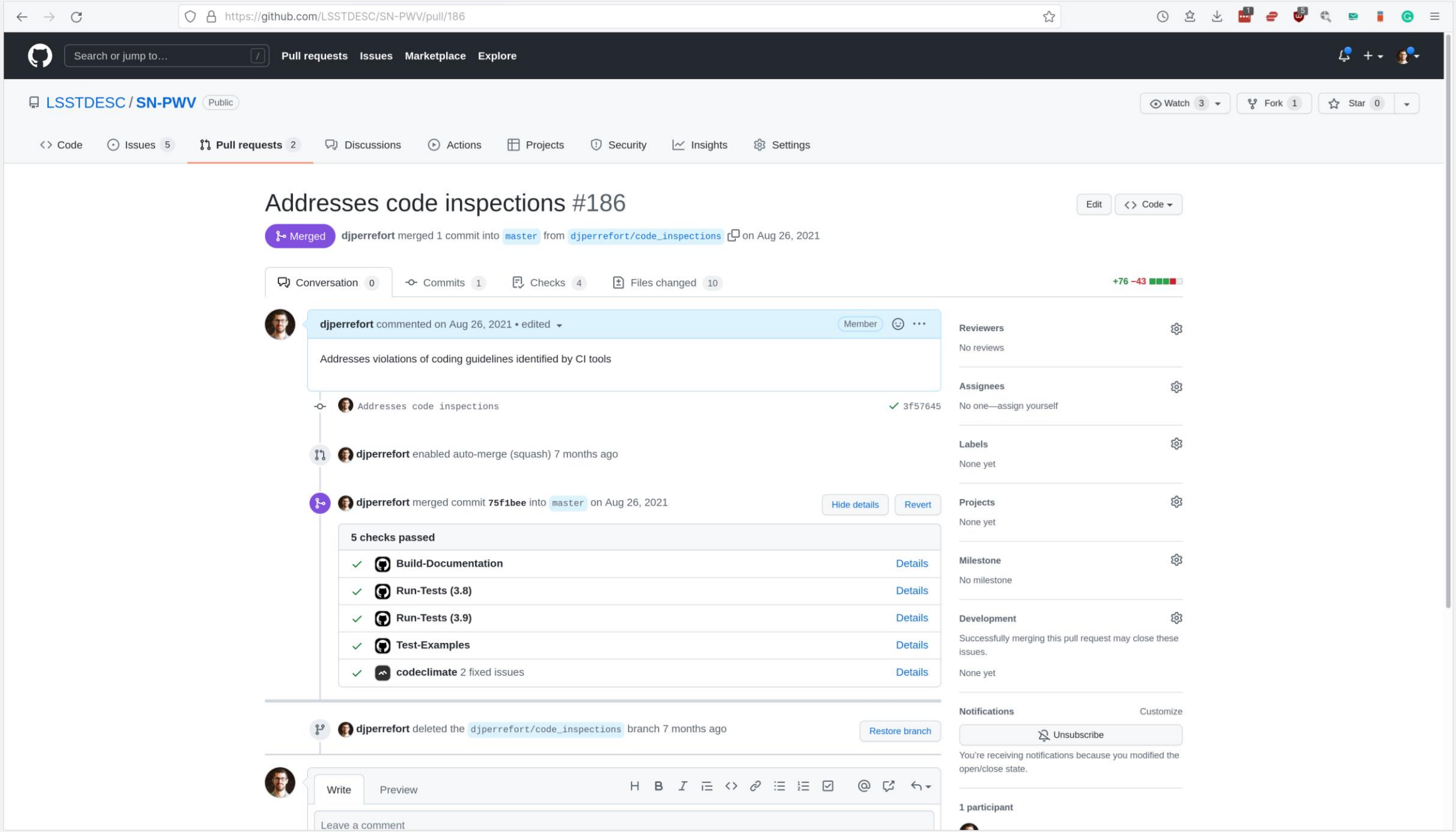
Requested changes from reviewer

Requested reviewers

Relevant issues


https://app.codacy.com/gh/mwvgroup/Pitt-Google-Broker/files?branch=master
mwvgroup
Pitt-Google-Broker
Community
Docs
Go back
Dashboard
Commits
Files
Issues
Pull Requests
Security
Code patterns
Settings
Pitt-Google-Broker
master
About this page
Search files
Missing some files?
Grade
Filename
Issues
Complexity
Duplication
Coverage
C broker / beam / bq_sink / bq_sink.py 3 issues 2 3 clones -
A broker / beam / bq_sink / setup.py 0 - 0 -
A broker / beam / __init__.py 0 - 0 -
C broker / beam / README.md 5 - - -
A broker / beam / value_added / setup.py 0 - 0 -
A broker / beam / value_added / transforms / filters.py 1 12 0 -
D broker / beam / value_added / transforms / __init__.py 1 - 0 -
A broker / beam / value_added / transforms / salt2.py 1 14 0 -
B broker / beam / value_added / value_added.py 5 2 4 -
A broker / broker_utils / broker_utils / beam_transforms.py 1 9 0 -
B broker / broker_utils / broker_utils / consumer_sim.py 9 6 0 -
A broker / broker_utils / broker_utils / data_utils.py 0 5 0 -


https://app.codacy.com/gh/mwvgroup/Pitt-Google-Broker/issues
mwvgroup
Pitt-Google-Broker
Community
Docs
Go back
Dashboard
Commits
Files
Issues
Pull Requests
Security
Code patterns
Settings
Current Issues
master
Take a tour
About this page
Filter
All languages
All categories
All levels
All patterns
All authors
broker/broker_utils/schema_maps/README.md
MINOR Code Style Expected: 80; Actual: 154
3 The files in this directory contain mappings between the schema of an individual survey and a PGB-standardized schema that is used within the broker code.
broker/cloud_functions/README.md
MINOR Code Style Expected: 80; Actual: 151
15 | `ps_to_gcs` | Listens to the `{survey}-alerts` Pub/Sub stream and stores each alert as an Avro file in Cloud Storage bucket `{survey}-alert_avros`. |
broker/cloud_functions/check_cue_response/README.md
MINOR Code Style Expected: 80; Actual: 98
3 This Cloud Function checks whether the broker responded appropriately to the auto-scheduler's cue.
MINOR Code Style Expected: 80; Actual: 133
4 It does this by first pausing to allow time for the response, and then checking each broker component, such as VMs and Dataflow jobs.
MINOR Code Style Expected: 80; Actual: 120
5 If a component is found to be in an unexpected state, a "Critical" error is raised which triggers a GCP alerting policy.
MINOR Code Style Expected: 1; Actual: 2
11
MINOR Code Style Expected: 80; Actual: 158
14 An alerting policy was created manually to notify Troy Raen of anything written to the log named `check-cue-response-cloudfnc` that has severity ``'CRITICAL'``.


https://github.com/LSSTDESC/SN-PWV/pull/186
Search or jump to…
Pull requests
Issues
Marketplace
Explore
LSSTDESC / SN-PWV
Public
Watch 3
Fork 1
Star 0
Code
Issues 5
Pull requests 2
Discussions
Actions
Projects
Security
Insights
Settings
Addresses code inspections #186
Edit
Code
Merged
djperrefort merged 1 commit into master from djperrefort/code_inspections on Aug 26, 2021
Conversation 0
Commits 1
Checks 4
Files changed 10
+76 −43
djperrefort commented on Aug 26, 2021 • edited
Member
Addresses violations of coding guidelines identified by CI tools
Addresses code inspections
3f57645
djperrefort enabled auto-merge (squash) 7 months ago
djperrefort merged commit 75f1bee into master on Aug 26, 2021
Hide details
Revert
5 checks passed
Build-Documentation
Details
Run-Tests (3.8)
Details
Run-Tests (3.9)
Details
Test-Examples
Details
codeclimate 2 fixed issues
Details
djperrefort deleted the djperrefort/code_inspections branch 7 months ago
Restore branch
Write
Preview
Leave a comment
Reviewers
No reviews
Assignees
No one—assign yourself
Labels
None yet
Projects
None yet
Milestone
No milestone
Development
Successfully merging this pull request may close these issues.
None yet
Notifications
Customize
Unsubscribe
You're receiving notifications because you modified the open/close state.
1 participant

Core Design Principles

- Big Design Up Front (BDUF)
- Keep It Simple (KISS)
- Principle of Least Surprise
- You Aren’t Going To Need It (YAGNI)
- Don't Repeat Yourself (DRY)

Object-Oriented Design (OOD)

- **S** - Single-responsibility Principle
- **O** - Open-closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle

Important Fundamentals	**PEP 8:** Style Guide for Python Code
	PEP 20: The Zen of Python
	PEP 257: Docstring Conventions
Bonus PEPs	**PEP 484:** Type Hints
	PEP 498: Literal String Interpolation
	PEP 572: Assignment Expressions