Refactoring Python Classes for NSLS-II Beamlines

Bryan Aleman, Elgin Community College, Elgin, IL 60123
Robert Schaffer, Jennefer Maldonado, National Synchrotron Light Source II, Brookhaven National Laboratory, Upton, NY, 11973

Abstract

The National Synchrotron Light Source II (NSLS-II) is one of the most advanced synchrotron light sources in the world, where many important scientific discoveries. With many of the beamlines offering unique experimental capabilities, researchers can probe materials, biological systems, and chemical processes at unparalleled resolution. Despite these differences, many beamlines rely on similar underlying software, which has led to duplication of code across instruments. This creates long-term maintenance and reliability challenges.

This project addresses that issue by refactoring and consolidating Ophyd classes, a Python library used to control devices at the beamlines. By unifying similar classes into a single, reusable, and maintainable package, the project enhances code readability, portability, and supportability. This work supports NSLS-II's mission to provide robust tools for scientific discovery. This project has also given me invaluable experience with software development in a collaborative setting, using industry-standard tools and preparing me for success in my academic and professional journey.

Introduction

NSLS-II operates dozens of beamlines that support a wide range of experiments in materials science, chemistry, biology, and engineering. Although the hardware and purpose of each beamline differ, they share a common software stack. Bluesky handles the orchestration of experiments, and Ophyd provides the device-level interfaces used to control motors, shutters, detectors, and other hardware.

Each beamline maintains its own profile-collection repository to define the devices and logic needed for its experiments. Over time, these repositories have evolved separately. As a result, many of the same kinds of Ophyd classes are re-implemented across beamlines with small variations. This duplication makes the codebase harder to maintain, harder to extend for new instrumentation, and less consistent across facilities that should behave in similar ways.

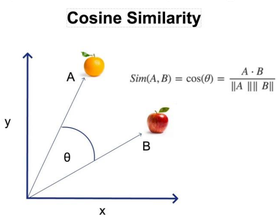
Addressing this duplication across many beamlines involves large volumes of code and is very time consuming to do with manual inspection. For that reason, this project uses automation-based approaches to identify, organize, and prepare these classes for refactoring.

Methods

To identify and prepare duplicated Ophyd classes for refactoring, two approaches were developed: one assisted by AI and one fully deterministic in Python.

First Attempt — AI-assisted clustering (Python + OpenAI API):

- Wrote a python script that extracted all Ophyd-related classes from profile-collection repositories **(700+ total classes)**, stored as JSON.
- Used **OpenAI API** to cluster classes by textual and structural similarity **(200+ clusters)** using cosine similarity (0.00–1.00 threshold).

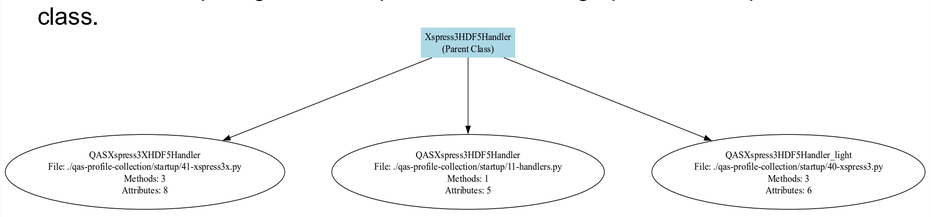



```python
def cosine_similarity(vec1, vec2):
    vec1 = np.array(vec1)
    vec2 = np.array(vec2)
    return float(np.dot(vec1, vec2) / (np.linalg.norm(vec1) * np.linalg.norm(vec2)))
```

- For each cluster, automatically generated a summary of the first three classes, noting shared patterns, differences, and initial refactoring suggestions (Markdown output).
- This produced fast insight but could not be trusted fully since only a subset of each cluster was reviewed.

Second Attempt — Deterministic structural analysis (no AI):

- Wrote a Python script to parse profile-collection repositories and extract Ophyd classes along with their parent relationships, attributes, and methods (JSON output).
- Built a second script to map class inheritance across all repositories and visualize relationships using **GraphViz**.
- Built a third script to generate separate inheritance graphs for each parent device class.

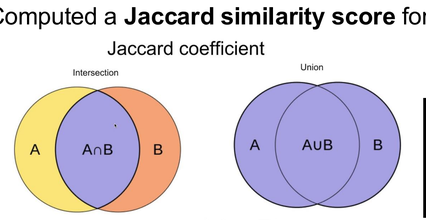


Similarity Detection:

- Developed a Python-based comparison method to evaluate true code similarity across child classes of the same parent.
- Defined similarity based on overlap in **attributes**, **methods**, and line-level code duplication.

> **Attributes**: Variables stored inside a class. Holds configuration or state (e.g. limits, channel names, device IDs).
> **Methods**: Functions defined inside a class. Control behavior or actions (e.g. move motor, read detector, trigger scan).

- Computed a **Jaccard similarity score** for each pair of child classes (JSON output).




```python
def compare_methods_and_attributes(class1, class2):
    def jaccard_similarity(set1, set2):
        intersection = len(set1 & set2)
        union = len(set1 | set2)
        return intersection / union if union != 0 else 0

    method_similarity = jaccard_similarity(set(class1["methods"]), set(class2["methods"]))
    attribute_similarity = jaccard_similarity(set(class1["attributes"]), set(class2["attributes"]))

    return method_similarity, attribute_similarity
```

- Classes with high attribute overlap, high method overlap, and high similarity score were flagged as candidates for consolidation and refactoring.

Outcomes

Using the AI-assisted approach, 781 Ophyd classes were extracted from the NSLS-II profile-collection repositories and grouped into 142 similarity clusters. The deterministic approach expanded this to 1,978 classes by parsing the repositories directly and recording each class's parent, children, attributes, and methods.

From these data, a global inheritance graph was generated across all repositories, followed by individual per-parent graphs to allow focused inspection. A Jaccard-based similarity report then compared every child class under each parent and produced structured scores for attribute overlap and method overlap. These results revealed multiple sets of highly similar classes implemented across different beamlines and identified a group of high-confidence candidates for refactoring. The produced artifacts : JSON reports, inheritance visualizations, and similarity score snippets. This now provides a reproducible basis for consolidation work moving forward.

The JSON snippet on the right shows the parent class Device and two child classes named Encoder, discovered in two different profile-collection repositories (QAS and ISS). The similarity scores indicate that these two Encoder definitions are nearly identical. The two code excerpts on the left and middle show the actual class definitions for these two children. Aside from a single extra line at the end of the ISS version, the implementations are effectively the same. This illustrates how the Jaccard report can guide targeted refactoring by pointing directly to duplicated classes across beamlines.


```
"Device": [
    {
        "class1": {
            "name": "Encoder",
            "file": "./qas-profile-collection/startup/10-detectors.py"
        },
        "class2": {
            "name": "Encoder",
            "file": "./iss-profile-collection/startup/30-detectors.py"
        },
        "method_similarity": 1.0,
        "attribute_similarity": 0.9285714285714286
    },
```


```python
class Encoder(Device):
    """This class defines components but does not implement actual reading.

    See EncoderFS and EncoderParser"""
    pos_I = Cpt(EpicsSignal, ')Cnt:Pos-I')
    sec_array = Cpt(EpicsSignal, ')T:sec_Bin_')
    nsec_array = Cpt(EpicsSignal, ')T:nsec_Bin_')
    pos_array = Cpt(EpicsSignal, ')Cnt:Pos_Bin_')
    index_array = Cpt(EpicsSignal, ')Cnt:Index_Bin_')
    data_array = Cpt(EpicsSignal, ')Data_Bin_')
    # The '$' in the PV allows us to write 40 chars instead of 20.
    filepath = Cpt(EpicsSignal, ')ID:File.VAL', string=True)
    dev_name = Cpt(EpicsSignal, ')DevName')

    filter_dy = Cpt(EpicsSignal, ')Fltr:dY-SP')
    filter_dt = Cpt(EpicsSignal, ')Fltr:dT-SP')
    reset_counts = Cpt(EpicsSignal, ')Rst-Cmd')

    ignore_rb = Cpt(EpicsSignal, ')Ignore-RB')
    ignore_sel = Cpt(EpicsSignal, ')Ignore-Sel')
```

```python
class Encoder(Device):
    """This class defines components but does not implement actual reading.

    See EncoderFS and EncoderParser"""
    pos_I = Cpt(EpicsSignal, ')Cnt:Pos-I')
    sec_array = Cpt(EpicsSignal, ')T:sec_Bin_')
    nsec_array = Cpt(EpicsSignal, ')T:nsec_Bin_')
    pos_array = Cpt(EpicsSignal, ')Cnt:Pos_Bin_')
    index_array = Cpt(EpicsSignal, ')Cnt:Index_Bin_')
    data_array = Cpt(EpicsSignal, ')Data_Bin_')
    # The '$' in the PV allows us to write 40 chars instead of 20.
    filepath = Cpt(EpicsSignal, ')ID:File.VAL', string=True)
    # filepath = Cpt(EpicsSignal, ')ID:File')
    dev_name = Cpt(EpicsSignal, ')DevName')

    filter_dy = Cpt(EpicsSignal, ')Fltr:dY-SP')
    filter_dt = Cpt(EpicsSignal, ')Fltr:dT-SP')
    reset_counts = Cpt(EpicsSignal, ')Rst-Cmd')

    ignore_sel = Cpt(EpicsSignal, suffix=')Ignore-RB', write_pv=')Ignore-Sel')
```

Discussion

The analysis shows that many NSLS-II beamlines maintain separate implementations of device classes that are functionally the same. Since all beamlines rely on Bluesky and Ophyd, this duplication reflects code being rewritten or copied instead of reused. This slows development and increases maintenance effort over time. By producing a complete structural map of Ophyd classes across profile-collection repositories, this work turns refactoring from a manual search problem into a targeted process.

These results matter because they make future beamline integration more efficient: instead of writing from scratch or copying code blindly, developers can now identify and reuse unified definitions. At the same time, the work exposed limitations: AI clustering was useful for exploration but not reliable for final decisions, and structural similarity alone does not capture intentional beamline-specific differences.

The next step is not finding duplication, that part is largely solved, but refactoring at scale. Because the number of candidate classes is large, doing this manually would be slow and error-prone. A practical path forward is to combine these outputs with controlled automation and LLM-assisted refactoring so that consolidation can be accelerated while still reviewed by humans for correctness.

Acknowledgements

I would like to thank my mentors Robert Schaffer and Jennefer Maldonado for their help on this project as well as AJ Silgar and Max Rakitin for offering guidance and support for this project.
This project was supported in part by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists (WDTS) under the Community College Internships Program (CCI). No export control.

Elgin Community College

U.S. DEPARTMENT of ENERGY

www.bnl.gov

Brookhaven National Laboratory