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 a synchrotron facility that generates extremely bright X-ray light for experimental use across disciplines in materials science, chemistry, biology, and engineering. Each beamline at NSLS-II maintains a separate software repository to control experimental devices using Ophyd, a Python library for hardware abstraction, and to orchestrate experiments using Bluesky. Over time, independent development of these repositories has resulted in duplication of Ophyd device classes, which increases the long-term cost of maintenance and slows the integration of new beamlines. In this work, this duplication was identified and characterized through two approaches: an AI-assisted clustering method using an embedding-based cosine similarity model, and a deterministic analysis that extracted class inheritance, attributes, and methods from 1,978 Ophyd classes across profile-collection repositories. Structural similarity among classes sharing a common parent was then quantified using a Jaccard index to identify candidates for consolidation. The results show that many device classes differ only by minor modifications, demonstrating that substantial refactoring is feasible. The outputs of this analysis, including inheritance maps, similarity scores, and explicit examples such as duplicated 'Encoder' classes across beamlines, provide a reproducible foundation for creating unified, maintainable Ophyd class definitions that support more efficient software development at NSLS-II.

I. Introduction

The National Synchrotron Light Source II (NSLS-II) is a large-scale research facility that provides extremely bright synchrotron X-ray light—up to ten billion times brighter than the sun[1]—for experiments in materials science, chemistry, biology, structural physics, and engineering. The facility operates dozens of beamlines, each designed for a specific class of methods such as hard X-ray diffraction, spectroscopy, imaging and microscopy, bio-imaging, electronic structure measurements, or complex scattering. Although the scientific goals and hardware configurations of these beamlines differ significantly, they all rely on common software tools to run experiments in a repeatable and controlled way.

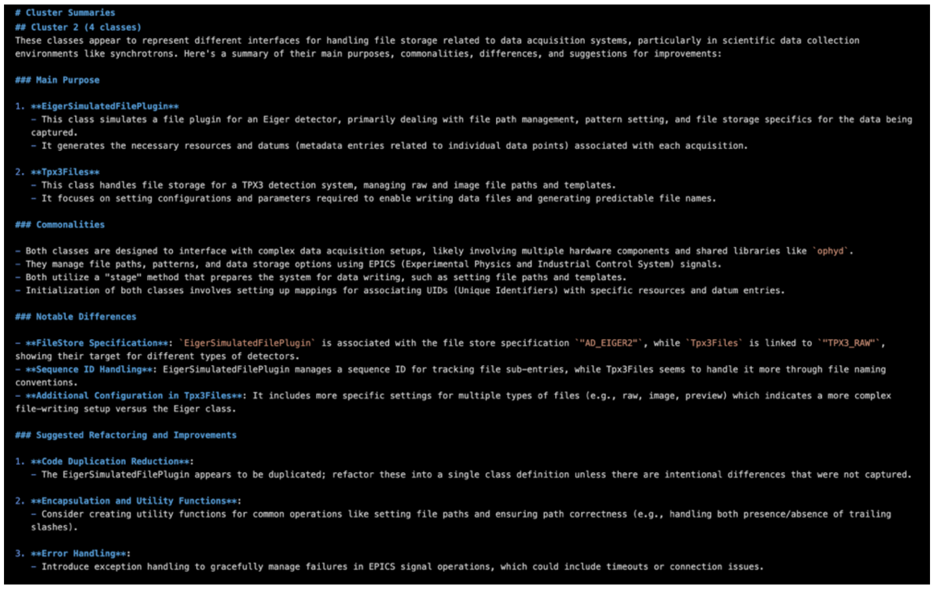
At NSLS-II, experiments are orchestrated using Bluesky[2], and device-level interaction is performed through Ophyd[3], a Python library for controlling motors, detectors, shutters, and other hardware components. Each beamline maintains its own "profile-collection" repository containing the Ophyd class definitions and other configuration logic needed for its operation. Because these repositories have evolved independently over time, many of the same device classes have been re-implemented multiple times with only minor differences. This duplication increases the effort required to maintain, review, and extend the software, and it slows down the integration of new beamlines.

The scope of this work is to identify and characterize duplicated Ophyd classes across NSLS-II profile-collection repositories, and to prepare a reproducible basis for refactoring them into unified definitions. The purpose is not to perform refactoring itself, but to create the structural information and similarity analysis needed to make consolidation efficient and technically justified.

II. Methods

A. AI-Assisted Clustering

In the first phase of the analysis, I wrote a Python script to extract 781 Ophyd-related classes from the NSLS-II profile-collection repositories. These classes were serialized to JSON and then converted into vector representations using the text-embedding-ada-002 model[5]. Cosine similarity was then applied to group classes by structural and textual similarity on a 0.00–1.00 scale, where values near 1.00 indicate stronger resemblance. This produced 142 clusters of related classes. For each cluster, I used the OpenAI API to generate a summary of the first three classes, highlighting shared patterns, differences, and potential refactoring directions.



```
# Cluster Summaries
## Cluster 2 (4 classes)
These classes appear to represent different interfaces for handling file storage related to data acquisition systems, particularly in scientific data collection
environments like synchrotrons. Here's a summary of their main purposes, commonalities, differences, and suggestions for improvements:

### Main Purpose

1. **EigerSimulatedFilePlugin**
   - This class simulates a file plugin for an Eiger detector, primarily dealing with file path management, pattern setting, and file storage specifics for the data being
   captured.
   - It generates the necessary resources and datums (metadata entries related to individual data points) associated with each acquisition.

2. **Tpx3Files**
   - This class handles file storage for a TPX3 detection system, managing raw and image file paths and templates.
   - It focuses on setting configurations and parameters required to enable writing data files and generating predictable file names.

### Commonalities

- Both classes are designed to interface with complex data acquisition setups, likely involving multiple hardware components and shared libraries like `ophyd`.
- They manage file paths, patterns, and data storage options using EPICS (Experimental Physics and Industrial Control System) signals.
- Both utilize a "stage" method that prepares the system for data writing, such as setting file paths and templates.
- Initialization of both classes involves setting up mappings for associating UIDs (Unique Identifiers) with specific resources and datum entries.

### Notable Differences

- **FileStore Specification**: `EigerSimulatedFilePlugin` is associated with the file store specification `"AD_EIGER2"`, while `Tpx3Files` is linked to `"TPX3_RAW"`,
showing their target for different types of detectors.
- **Sequence ID Handling**: EigerSimulatedFilePlugin manages a sequence ID for tracking file sub-entries, while Tpx3Files seems to handle it more through file naming
conventions.
- **Additional Configuration in Tpx3Files**: It includes more specific settings for multiple types of files (e.g., raw, image, preview) which indicates a more complex
file-writing setup versus the Eiger class.

### Suggested Refactoring and Improvements

1. **Code Duplication Reduction**:
   - The EigerSimulatedFilePlugin appears to be duplicated; refactor these into a single class definition unless there are intentional differences that were not captured.

2. **Encapsulation and Utility Functions**:
   - Consider creating utility functions for common operations like setting file paths and ensuring path correctness (e.g., handling both presence/absence of trailing
   slashes).

3. **Error Handling**:
   - Introduce exception handling to gracefully manage failures in EPICS signal operations, which could include timeouts or connection issues.
```

Figure 1. AI clustering summary

Although this clustering provided a fast preview of duplication, it evaluated only a subset of each cluster and could not be relied on as a definitive basis for refactoring. It was therefore used primarily as a reconnaissance step to understand the scale and distribution of similarity across beamlines.

B. Deterministic Structural Analysis

To produce a fully reproducible and auditable dataset, I developed a second pipeline that did not depend on AI. A Python script was written to scan all profile-collection repositories and extract Ophyd classes along with their parent class, attributes, and methods. In this context, attributes are variables stored inside the class that hold configuration or state, such as channel names, device IDs, or motor limits. Methods are the functions inside a class that define its behavior, such as move(), read(), trigger(), or configure().

This stage extracted 1,978 classes and constructed a global inheritance map across all repositories using GraphViz[4]. Because the full inheritance graph was extremely dense, I generated additional per-parent graphs to isolate and visualize the children of specific device classes.

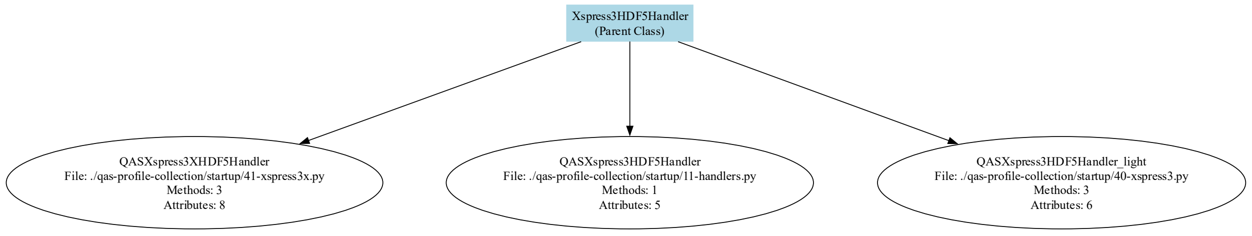
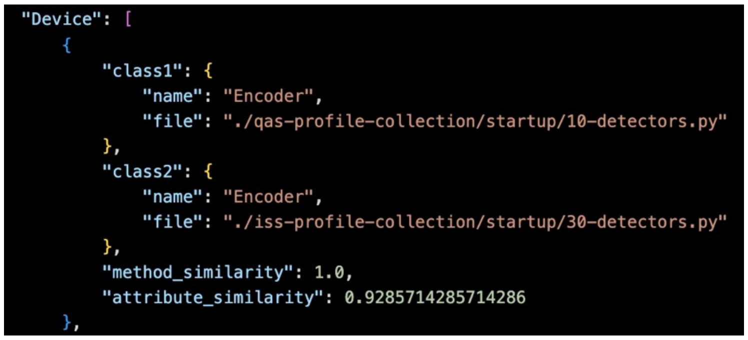

Xspress3HDF5Handler
(Parent Class)
QASXspress3XHDF5Handler
File: ./qas-profile-collection/startup/41-xspress3x.py
Methods: 3
Attributes: 8
QASXspress3HDF5Handler
File: ./qas-profile-collection/startup/11-handlers.py
Methods: 1
Attributes: 5
QASXspress3HDF5Handler_light
File: ./qas-profile-collection/startup/40-xspress3.py
Methods: 3
Attributes: 6

Figure 2. Per-parent graphs

C. Similarity Computation and Candidate Identification

To quantify duplication among children of the same parent class, I computed pairwise Jaccard similarity scores using the sets of attributes and methods extracted from each class. In this context, the Jaccard score expresses how much two classes overlap relative to the total amount of content they contain. A score is higher when most of the attributes and methods are shared, and lower when they differ. Classes with high overlap and minimal differences were flagged as likely duplication cases and became candidates for refactoring in later stages of the project.



```
"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
    },
```

Figure 3. JSON Jaccard similarity report for parent class 'Device' and its children

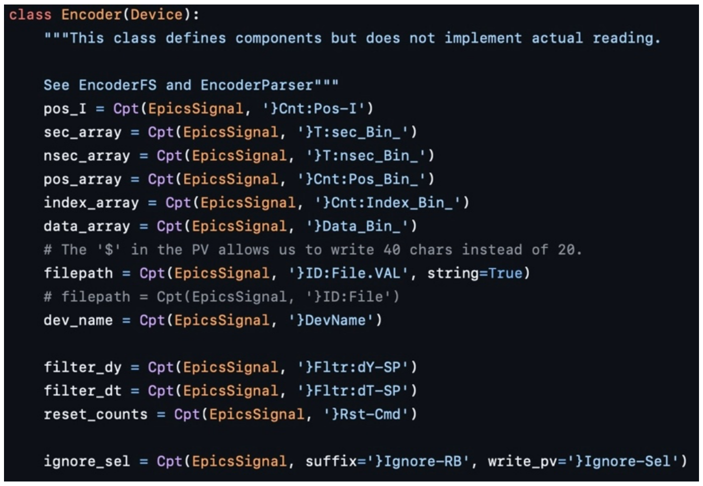
III. Results

A total of 781 Ophyd classes were extracted from the NSLS-II profile-collection repositories during the AI-assisted phase and grouped into 142 similarity clusters. Summary reports were generated for each cluster based on the first three members.

In the deterministic phase, 1,978 classes were collected by parsing the repositories directly. For each class, the parent class, list of attributes, and list of methods were recorded. Using these data, a global inheritance map was generated across all repositories, followed by a set of per-parent inheritance graphs to isolate structural relationships for individual device classes.

Pairwise Jaccard similarity was computed for child classes under each parent class using the extracted attributes and methods. The results were stored as a JSON report. Classes with high overlap were identified and marked as candidates for refactoring.

As an example, two implementations of the 'Encoder' class were found in the ISS (Figure 4) and QAS (Figure 5) profile-collection repositories. The Jaccard similarity report in Figure 3 showed a high overlap, and inspection of the source code confirmed that the two implementations differed only by a single line.



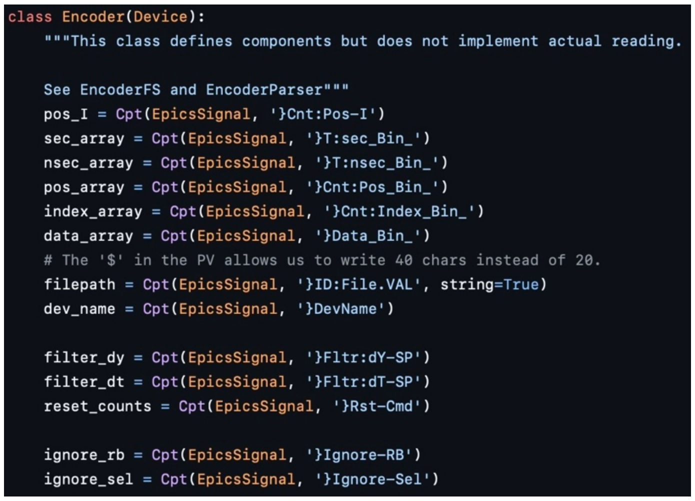
```python
class Encoder(Device):
    """This class defines components but does not implement actual reading.

    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')
```

Figure 4. ‘Encoder’ class in ISS profile-collection files that inherits from ‘Device’



```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')
```

Figure 5. ‘Encoder’ class in QAS profile-collection files that inherits from ‘Device’

IV. Discussion

The results show that multiple NSLS-II beamlines have developed functionally similar Ophyd device classes independently. The presence of high-similarity pairs across different repositories indicates that identical or near-identical code has been recreated in multiple locations rather than reused from a shared source. The large number of duplicated implementations suggests that this pattern is systemic rather than incidental. The availability of complete structural maps and similarity measurements confirms that the duplication is both measurable and traceable to specific repositories and device types. The example of the duplicated Encoder class across ISS and QAS illustrates that the observed similarity is not limited to superficial naming conventions but extends to the underlying structure of attributes and methods.

V. Conclusion

The analysis conducted in this project demonstrates that duplication of Ophyd device classes exists across the NSLS-II profile-collection repositories. Multiple beamlines have developed independent implementations of the same or nearly identical classes, a pattern confirmed quantitatively through structural extraction and similarity scoring, and qualitatively through source inspection. The duplication is not limited to minor naming differences but extends to shared internal structure, indicating that parallel development has occurred repeatedly over time.

By generating a complete inventory of Ophyd classes, mapping inheritance relationships, and computing structured similarity metrics, the work establishes a reproducible foundation for addressing this duplication. The availability of explicit candidates for consolidation removes the need for manual searching and transforms refactoring from an open-ended task into a directed engineering effort. The example of the ‘Encoder’ class illustrates the practical relevance of the analysis and shows how similarity-based identification can guide targeted consolidation.

Based on these findings, further work should focus on applying controlled refactoring to the identified high-similarity classes, with human verification to ensure that beamline-specific behavior is preserved where necessary. Because the number of candidates is large, a practical strategy for refactoring should incorporate automation or large-language-model assistance to accelerate rewriting while retaining developer oversight. In addition, the structural and similarity analysis developed here may be run periodically to prevent the re-introduction of duplication as new beamlines are added or existing ones evolve.

This work supports the maintainability and consistency of the NSLS-II software ecosystem and provides a structured reference point for future consolidation efforts across beamlines.

VI. 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).

VII. References

[1]NSLS-II About Page, Brookhaven National Laboratory, https://www.bnl.gov/nsls2/about-nsls-ii.php (accessed 2025).

[2]Bluesky Project Documentation, Bluesky Project, https://blueskyproject.io (accessed 2025).

[3]Ophyd Documentation, Bluesky Project, https://blueskyproject.io/ophyd (accessed 2025).

[4]GraphViz Documentation, https://graphviz.org (accessed 2025).

[5]OpenAI, text-embedding-ada-002 model, https://platform.openai.com (accessed 2025).

VIII. Appendix

A. Participants

Name	Role	Institution / Affiliation
Bryan Aleman	Intern — Conducted analysis of duplicated Ophyd classes and generated supporting outputs	Elgin Community College
Robert Schaffer	Mentor — Provided technical guidance and project supervision	NSLS-II, DSSI / DAD
Jennefer Maldonado	Mentor — Provided technical guidance and project supervision	NSLS-II, DSSI / DAD

B. Scientific Facilities

This work was performed at the National Synchrotron Light Source II (NSLS-II), a U.S. Department of Energy Office of Science User Facility located at Brookhaven National Laboratory.