Refactoring Python Classes for NSLS-II Beamlines

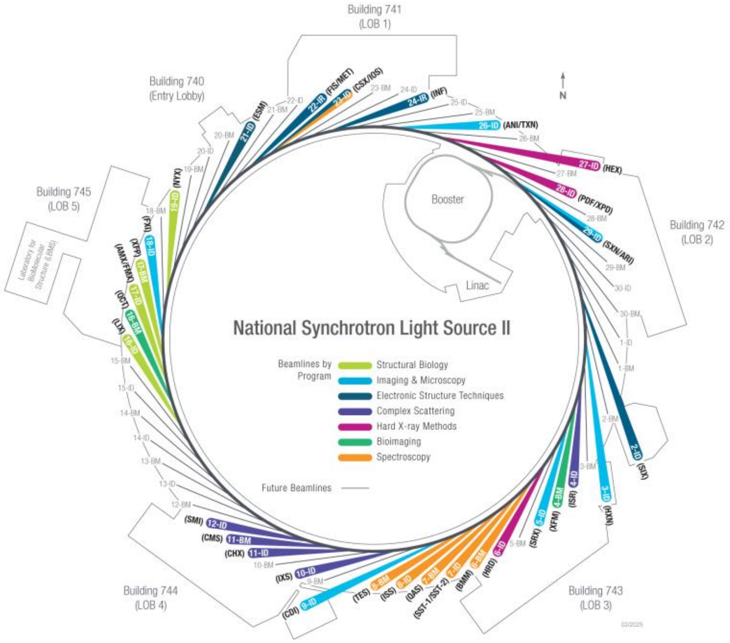
Bryan Aleman
Elgin Community College

Robert Schaffer, Jennefer Maldonado
National Synchrotron Light Source II, Brookhaven National Laboratory

October 29, 2025

Background

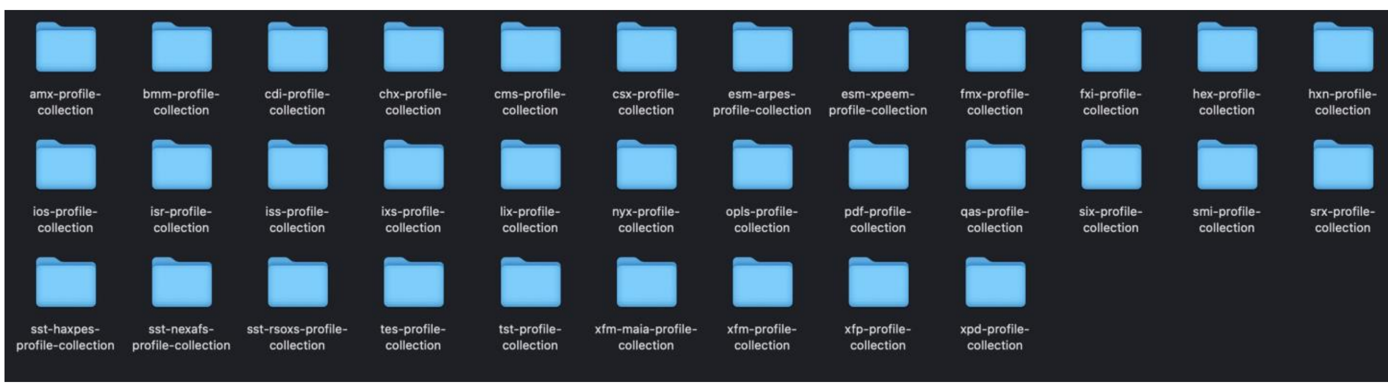
- NSLS-II enables scientific experiments across many disciplines using specialized beamlines.
- Beamlines differ in hardware and purpose but rely on the same software stack (Bluesky + Ophyd).
- Each beamline maintains its own code, leading to repeated implementations of similar device classes.



Brookhaven National Laboratory

Why This Matters

- Rewriting or copying code increases development time and maintenance cost.
- Duplicated classes drift over time and behave inconsistently across beamlines.
- Manual refactoring across dozens of repositories is not scalable.

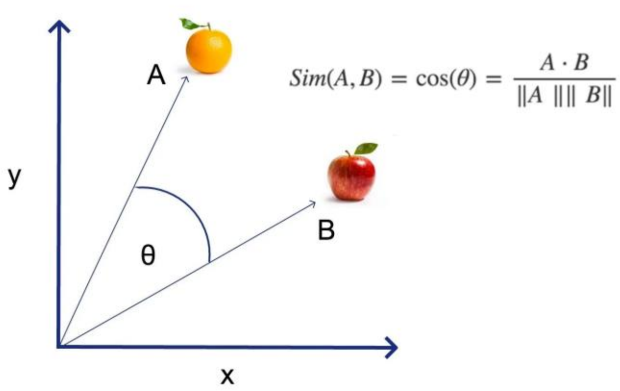


Brookhaven National Laboratory

Methods: AI-Assisted Attempt

- Extracted 781 Ophyd classes from profile-collection repositories (JSON output).
- Clustered into 142 groups using OpenAI API with cosine similarity.

Cosine Similarity




```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)))
```

Brookhaven National Laboratory

Methods: AI-Assisted Attempt

- Generated summaries for common patterns and initial refactoring suggestions.
- Fast for exploration, but not fully reliable for refactoring decisions.

Brookhaven National Laboratory

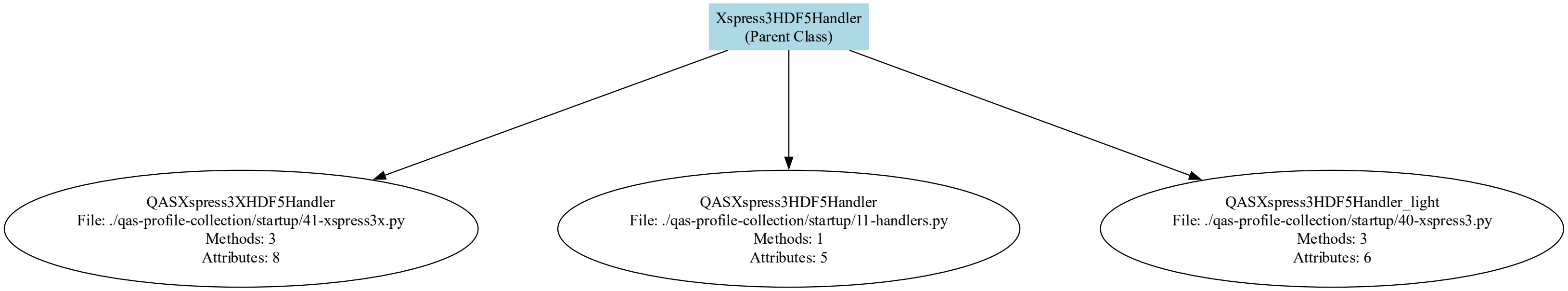
Methods: AI-Assisted Attempt

```
# 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.
```

Brookhaven National Laboratory

Methods: Deterministic Analysis

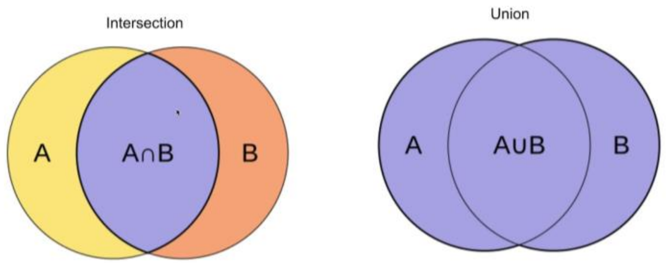
- Parsed repositories again and found 1,978 classes with parents, attributes, and methods.
- Built a global inheritance graph and per-parent graphs using GraphViz.



Methods: Deterministic Analysis

- Computed Jaccard similarity across child classes to detect structural duplication.
- High-similarity pairs flagged as refactoring candidates.

Jaccard coefficient



$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

```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
```

Brookhaven National Laboratory

Outcomes

- Mapped full class structure across all NSLS-II profile-collection repositories.
- Identified multiple sets of near-identical classes across different beamlines.
- Produced JSON reports, graphs, and similarity scores to guide refactoring work.

Outcomes

- Example: JSON similarity report highlighting ‘device’ parent class and their children both named ‘Encoder’ for QAS and ISS beamline.


```
"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
    },
```


Brookhaven National Laboratory

Outcomes

- Example: The ‘Encoder’ classes from QAS and ISS differ by one line here.

QAS

```
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')
```

ISS

```
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')
```

Brookhaven National Laboratory

Summary

- Code duplication across beamlines is real, widespread, and now quantifiable.
- The analysis converts refactoring from manual search into a targeted process.
- These findings support creating shared, reusable class definitions to improve long-term maintainability.

Next Steps

- Consolidation will reduce maintenance cost and improve consistency for future beamlines.
- Manual refactoring alone is too slow given the number of candidates.
- Next phase: use controlled automation and LLM-assisted refactoring with human review to accelerate this work safely.

Thank you!