Winnow: Software design specification

Ayan Das

March 9, 2025

Contents

1 Introduction **2**
- 1.1 Purpose and Scope . . . 2
- 1.2 Relationship to Other Documents . . . 3
- 1.3 Document Conventions . . . 3

2 Detailed Design Objectives **3**
- 2.1 Mapping to Requirements . . . 3

3 System/Module Decomposition **4**
- 3.1 Module `InputParser` . . . 4
- 3.2 Module `AbstractAugmenter` . . . 5
- 3.3 Module `FilterEngine` . . . 5
- 3.4 Module `Exporter` . . . 6
- 3.5 Module `Plugin Infrastructure` . . . 7
- 3.6 Module `Shared / Utilities` . . . 7

4 Data Design **7**
- 4.1 Reference Data Structure . . . 7
- 4.2 Configuration Data Structure . . . 8
- 4.3 Data Flow Within Modules . . . 8
- 4.4 Validation and Transformation Logic . . . 9

5 Interface Design **9**
- 5.1 Internal Interfaces Between Components . . . 9
- 5.2 Method Signatures (Detailed) . . . 9

6 Behavior and Control Flows **11**
- 6.1 High-Level Pipeline Sequence (Activity Diagram) . . . 11
- 6.2 Key Algorithms . . . 11
  - 6.2.1 Lexicon Intersection Algorithm . . . 11
  - 6.2.2 Classifier-Based Filtering . . . 12
  - 6.2.3 AbstractAugmenter Fallback Logic . . . 12

7 Error Handling, Logging, and Monitoring **12**
7.1 Exception Handling Strategy 12
7.2 Logging Approach 13
7.3 Monitoring 13

8 Security and Access Control **13**
8.1 API Keys and Tokens 13
8.2 HTTPS for External Calls 13
8.3 Local File Permissions 13

9 Performance Considerations **14**
9.1 Profiling Strategy 14
9.2 Caching and Bulk Operations 14
9.3 Concurrency Approaches 14

10 Testing and Quality Assurance **14**
10.1 Unit Testing 14
10.2 Integration Testing 15
10.3 Code Review and Static Analysis 15
10.4 Performance Testing 15

11 Implementation Plan **15**
11.1 Step-by-Step Guidance 15
11.2 Version Control Guidelines 16

12 Appendices **17**
12.1 Potential Data Structures for Multilingual Support (Future Work) 17
12.2 Sample Config File 17
12.3 References to Other Docs 17

1 Introduction

1.1 Purpose and Scope

This **Software Design Specification (SDS)** provides a detailed, *implementation-level* view of the system architecture described in the *Software Architecture Specification (SAS).* While the SAS covered high-level modules, concurrency, and interfaces, this SDS focuses on:

- The *internal structure and design* of each module (with classes, functions, data flows).
- *Key algorithms* for abstract augmentation, filtering, and text classification.
- *Detailed data models* and method signatures.
- The *error-handling, logging, testing,* and *implementation plan.*

1.2 Relationship to Other Documents

- **SRS (Software Requirements Specification):** Defines the functional and non-functional requirements.
- **SAS (Software Architecture Specification):** Outlines the high-level structure, modules, concurrency, and data flow.
- **This SDS:** Provides a thorough *software-level design*, bridging the conceptual architecture and the actual Python code.

1.3 Document Conventions

- This SDS uses `typewriter font` to denote code or function signatures.
- Pseudo-UML diagrams are textual for illustrative purposes.
- Some code snippets are given in Python-like pseudocode to show design-level details.

2 Detailed Design Objectives

2.1 Mapping to Requirements

The main objectives trace to both functional and non-functional requirements in the SRS:

FR-1: Parse & Validate BibTeX files from directories.

FR-2: Apply configurable invariants.

FR-3: Augment missing abstracts from external services.

FR-4: Filter references based on domain lexicons (Physics & ML).

FR-5: (Optional) Use advanced text classification to refine the candidate set.

FR-6: Provide a plugin interface for specialized scrapers.

FR-7: Robust logging, error handling.

FR-8: Export final references to curated '.bib' files.

Non-functional goals such as **scalability**, **maintainability**, **performance**, and **extensibility** guide the design approach, ensuring that each module has clearly delimited responsibilities and a well-defined interface.

3 System/Module Decomposition

Following the *Logical View* from the SAS, we implement a **pipeline of modules**:

I. **Module: InputParser**

II. **Module: AbstractAugmenter**

III. **Module: FilterEngine**

IV. **Module: Exporter**

V. **Module: Plugin Infrastructure (Scrapers, Classifiers)**

VI. **Module: Shared / Utilities (Configuration, Logging, Exceptions)**

Below, each module is broken down into subcomponents/classes, responsibilities, and key methods.

3.1 Module `InputParser`

Responsibilities:

- Recursively discover '.bib' files under a given directory.
- Parse each '.bib' entry into an in-memory `Reference` object.
- Apply basic data invariants (e.g., no empty `title`).
- Yield or return a list of `Reference` objects for further processing.

Classes/Functions Overview:

`Reference` (Data Class)
- Fields: `bibtex_key`, `title`, `authors`, `year`, `abstract`, `venue`, `doi`, `url`, `raw_bibtex`.
- `__init__`(...) constructs a Reference from given fields.

`BibFileParser`
- `parse_file(file_path: str) -> List[Reference]`: Uses an external library (e.g., `bibtexparser`) to parse a single '.bib'.

`InputParser` (the main orchestrator)
- `discover_bib_files(root_dir: str) -> List[str]`: Recursively find '.bib' files.
- `parse_references(root_dir: str, config: Config) -> Iterator[Reference]`:
  - For each '.bib' file discovered, parse it into `Reference` objects.
  - Apply invariants (from `config.invariants`).
  - Log or skip references that fail invariants.
  - Yield valid references.

3.2 Module `AbstractAugmenter`

Responsibilities:

- Identify references lacking an `abstract`.
- Query external services (Semantic Scholar (SS), OpenAlex (OA), arXiv, Google Scholar) in a configurable fallback order.
- Possibly run specialized `ScraperPlugins` (e.g., for *Nature* or *Physical Review E*).
- Update the `abstract` field if a match is found.

Classes/Functions Overview:

`AbstractScraper` (Base Class)
- `get_abstract(ref: Reference) -> Optional[str]`
  - Takes a `Reference` and attempts to fetch an abstract via whichever method the subclass implements.
  - Returns `None` if the scraper cannot find a valid abstract.

`SSScraper`, `OAScraper`, `ArxivScraper`, etc.
- Implement `get_abstract()` for each external service.
- Typically do an HTTP request with `requests` or an equivalent library.
- Parse the JSON/XML response, return `abstract` if found.

`AbstractAugmenter`
- Fields:
  - `scrapers: List[AbstractScraper]`
  - `config: Config`
  - `cache: dict` (optional) for storing `(title,abstract)` pairs.
- `augment(refs: Iterable[Reference]) -> None`:
  - Iterates over `refs`.
  - For each ref without an abstract:
    1. Check local `cache`.
    2. If not in cache, loop through the `scrapers` in order:
       * `potential_abstract = scraper.get_abstract(ref)`
       * If `potential_abstract` is not `None`, set `ref.abstract` and break.
    3. Update the cache accordingly.

3.3 Module `FilterEngine`

Responsibilities:

- *Lexicon-Based Filtering*: Check if `(title + abstract)` contains at least one physics term and at least one ML term.

- *(Optional) Classifier-Based Filtering*: Use a trained text classifier to refine the candidate set and reduce false positives.
- Output: A final list of *candidate* references.

Classes/Functions Overview:

`LexiconFilter`
- `__init__`(physics_lexicon: List[str], ml_lexicon: List[str], config: Config)
- `matches_intersection(text: str) -> bool`:
  - Implements partial or regex matching for terms in both lexicons.
  - Returns `True` if *at least one* physics term and *at least one* ML term is found in the string `text`.

`ClassifierFilter` (Optional)
- `__init__`(model: SomeModel, config: Config)
- `predict_relevance(text: str) -> float`:
  - Returns a probability or confidence of relevance to the *physics+ML* intersection.
- `is_relevant(prob: float) -> bool`:
  - Compares the probability to a threshold from `config.classifier_threshold`.

`FilterEngine`
- `__init__`(lexicon_filter: LexiconFilter, classifier_filter: Optional[ClassifierFilter], config: Config)
- `filter_refs(refs: Iterable[Reference]) -> List[Reference]`:
  - For each reference:
    1. Build `text = (ref.title + " " + ref.abstract).lower()`.
    2. `if lexicon_filter.matches_intersection(text) == False: exclude it.` (Stop if not included, or set a *candidate* flag to `False`.)
    3. If the classifier is enabled:
       (a) `prob = classifier_filter.predict_relevance(text)`
       (b) If `classifier_filter.is_relevant(prob) == False`, exclude it.
    4. If still included, keep it in the final list.
  - Return the final candidate list.

3.4 Module `Exporter`

Responsibilities:

- Receive the filtered references and write them to a single (or multiple) output '.bib' file(s).
- Optionally include additional fields (like classification confidence, or custom keywords) if desired.

Classes/Functions Overview:

`BibExporter`
- `__init__`(config: Config)
- `export_bib(refs: List[Reference], output_path: str) -> None`:
  - Iterates over `refs` and reconstructs each entry into a valid BibTeX string (possibly using `bibtexparser` or `pybtex`).
  - Ensures that newly added `abstract` fields are included.
  - Writes the final '.bib' to disk.

3.5 Module `Plugin Infrastructure`

Responsibilities:

- Provide base classes or interfaces for `ScraperPlugins` and `ClassifierPlugins`.
- Dynamically load plugins based on user configuration.

Classes/Functions Overview:

`PluginLoader`
- `load_scrapers(plugin_paths: List[str]) -> List[AbstractScraper]`:
  - Dynamically import modules from the specified paths, ensure they implement `AbstractScraper`.
  - Return a list of instantiated scraper objects.
- `load_classifier(plugin_path: str) -> ClassifierFilter`:
  - Similarly, load a classifier plugin from a Python module path.

3.6 Module `Shared / Utilities`

- **Configuration Management:** Parsing YAML/JSON config for lexicon paths, concurrency, API keys, thresholds, etc.
- **Logging Infrastructure:** Possibly a wrapper around Python's `logging` module to produce structured logs (e.g., JSON logs).
- **Exception Definitions:** E.g., `AugmentationError`, `ParsingError`, etc.

4 Data Design

4.1 Reference Data Structure

```
@dataclass
class Reference:
    bibtex_key: str
    title: str
```

```
    authors: str
    year: str
    abstract: Optional[str] = None
    venue: Optional[str] = None
    doi: Optional[str] = None
    url: Optional[str] = None
    raw_bibtex: str = ""   # store the original unmodified entry
```

Notes:

- Some fields may be absent in certain references.
- `authors` can be a single string or a list of strings. The `raw_bibtex` is kept for output fidelity.

4.2 Configuration Data Structure

Typically loaded from `config.yaml` or `config.json`:

```
@dataclass
class Config:
    root_dir: str
    output_file: str
    invariants: Dict[str, Any]  # e.g. {"require_title": True, "require_year": False}
    lexicon_paths: List[str]    # paths to 'physics.txt', 'ml.txt'
    concurrency: str            # e.g. "thread", "process", or "none"
    classifier_enabled: bool
    classifier_threshold: float
    # etc...

    # External services configuration
    semantic_scholar_api_key: Optional[str] = None
    openalex_api_key: Optional[str] = None
    # ...
```

4.3 Data Flow Within Modules

1. **InputParser** -> yields `Reference` objects.
2. **AbstractAugmenter** -> modifies `Reference.abstract` in place if found.
3. **FilterEngine** -> reads the fields, decides keep/exclude.
4. **Exporter** -> compiles final '.bib' strings from the `Reference` objects that pass the filter.

4.4 Validation and Transformation Logic

- **Invariants Check:**
  - `require_title`: If `title` is empty, skip or log an error.
  - `require_year`: If `year` is missing, assign `"unknown"` or skip.
  - `max_authors_check`: If `authors` is extremely large or ill-formatted, log a warning.
- **Lexicon Normalization:**
  - Lexicon files can contain lines with `Ising`, `Spin glass`, etc.
  - The `FilterEngine` loads them into memory (list or set) and may pre-compile them into regex patterns.

5 Interface Design

5.1 Internal Interfaces Between Components

1 **InputParser -> AbstractAugmenter:**

```
references = InputParser().parse_references(root_dir, config)
AbstractAugmenter(scrapers, config).augment(references)
```

The `references` is a list or iterator of `Reference` objects.

2 **AbstractAugmenter -> FilterEngine:**

```
filtered_refs = FilterEngine(lexicon_filter, classifier_filter, config)
                .filter_refs(references)
```

3 **FilterEngine -> Exporter:**

```
BibExporter(config).export_bib(filtered_refs, config.output_file)
```

5.2 Method Signatures (Detailed)

The following are *key* methods in a simplified signature format:

```python
class InputParser:
    def parse_references(self, root_dir: str, config: Config) -> List[
       Reference]:
        """Discover .bib files, parse them, apply invariants, return
            references."""
        ...

class AbstractAugmenter:
    def __init__(self, scrapers: List[AbstractScraper], config: Config):
        self.scrapers = scrapers
        self.config = config
        self.cache = {}

    def augment(self, refs: Iterable[Reference]) -> None:
        """Iterate over refs, for each missing abstract, call scrapers in
            sequence, update abstract."""

class LexiconFilter:
    def __init__(self, physics_lexicon: List[str], ml_lexicon: List[str],
       config: Config):
        self.physics_lexicon = physics_lexicon
        self.ml_lexicon = ml_lexicon
        self.config = config
        # Possibly pre-compile regex patterns

    def matches_intersection(self, text: str) -> bool:
        """Return True if text has >=1 physics term AND >=1 ML term."""
        ...

class ClassifierFilter:
    def __init__(self, model_path: str, config: Config):
        self.model = self.load_model(model_path)
        self.config = config

    def load_model(self, model_path: str) -> object:
        """Load a pre-trained model from disk or memory."""
        ...

    def predict_relevance(self, text: str) -> float:
        """Return a probability [0.0..1.0]."""
        ...

    def is_relevant(self, prob: float) -> bool:
        return prob >= self.config.classifier_threshold

class FilterEngine:
    def __init__(self, lexicon_filter: LexiconFilter,
                 classifier_filter: Optional[ClassifierFilter],
                 config: Config):
        self.lex_filter = lexicon_filter
        self.clf_filter = classifier_filter
        self.config = config

    def filter_refs(self, refs: Iterable[Reference]) -> List[Reference]:
```

```
        """Return a list of references that pass lexicon (and classifier if
           enabled)."""
        ...

class BibExporter:
    def __init__(self, config: Config):
        self.config = config

    def export_bib(self, refs: List[Reference], output_path: str) -> None:
        """Write references to .bib file, preserving raw bibtex or updating
           fields as needed."""
        ...
```

6 Behavior and Control Flows

6.1 High-Level Pipeline Sequence (Activity Diagram)

Step 1: Parse 1.1: *InputParser.parse_references(root_dir, config)*
1.2: For each discovered '.bib', parse entries → `Reference` objects.
1.3: Validate each reference, log or skip if invariants fail.
↓
Step 2: Augment 2.1: *AbstractAugmenter.augment(references)*
2.2: For each ref without abstract, call scrapers in sequence.
2.3: If a scraper returns an abstract, store it and break.
↓
Step 3: Filter 3.1: *FilterEngine.filter_refs(references)*
3.2: *LexiconFilter.matches_intersection(text)*: pass/fail.
3.3: If classifier enabled, *ClassifierFilter.predict_relevance* & *is_relevant*.
3.4: Produce final candidate set.
↓
Step 4: Export 4.1: *BibExporter.export_bib(filtered_refs, config.output_file).*
4.2: Generate final '.bib' string, write to disk.

6.2 Key Algorithms

6.2.1 Lexicon Intersection Algorithm

1. Convert `(title + abstract)` to lowercase `text`.
2. Initialize `found_physics = False, found_ml = False`.
3. For each token in `physics_lexicon`:
   - If `token` is in `text` (substring or regex match), `found_physics = True`, break.
4. For each token in `ml_lexicon`:

- If `token` is in `text`, `found_ml = True`, break.

5. Return `found_physics && found_ml`.

6.2.2 Classifier-Based Filtering

1. `prob = model.predict_relevance(text)` returns a float $[0, 1]$.
2. Compare with `config.classifier_threshold`, typically 0.5 or 0.7.
3. Keep the reference if `prob` >= threshold, else discard.

6.2.3 AbstractAugmenter Fallback Logic

1. Maintain a list `scrapers` in desired priority, e.g. `[SemanticScholarScraper, OpenAlexScraper, ArxivScraper, GoogleScholarScraper]`.
2. For each `ref` lacking `abstract`:
   (a) If `ref.title` or `ref.doi` is found in `cache`, use it immediately.
   (b) Otherwise:
      - For each `scraper` in `scrapers`:
        i. `abstract = scraper.get_abstract(ref)`
        ii. If not `None`, set `ref.abstract = abstract`, store in cache, break.
   (c) If no scraper succeeds, `ref.abstract` remains `None`.

7 Error Handling, Logging, and Monitoring

7.1 Exception Handling Strategy

- **Parsing Exceptions**: If a '.bib' file is malformed, `BibFileParser` may throw `ParsingError`. We *catch* it in `InputParser.parse_references`, log the filename, and continue with the next file.
- **Augmentation Exceptions**:
  - If a scraper fails (e.g., network error, rate limit), it throws `AugmentationError` or returns `None`. The `AbstractAugmenter` logs this error, tries the next scraper.
  - *Partial success* is always favored over halting the pipeline.
- **Filtering Exceptions**: Typically none, but if the `ClassifierFilter` model is missing or corrupted, we log a warning and degrade to *lexicon-only* filtering if feasible.
- **Export Exceptions**: If writing the output file fails, we log an `ExportError` with the path. Possibly attempt writing to an alternative file or a temporary file.

7.2 Logging Approach

- Use Python's built-in `logging` module:
  - `logging.info("Starting parse for file: %s", file_path)`
  - `logging.error("Augmentation failed for reference: %s, reason: %s", ref.bibtex_k e)`
- Log levels: DEBUG, INFO, WARNING, ERROR, CRITICAL
- **Optional JSON Logs**: The system can be configured to produce JSON logs for better machine parsing.

7.3 Monitoring

- If run on a server, logs can be centralized via standard logging frameworks (`ELK`, `Graylog`, etc.).
- The system can produce a *summary report* at the end with:
  - Number of references parsed
  - Number of references with newly found abstracts
  - Number of references passing the filter

8 Security and Access Control

8.1 API Keys and Tokens

- The code must *never* print or log API keys in plaintext.
- Keys are stored in the `Config` object, read from environment variables or a secure config file.

8.2 HTTPS for External Calls

- All scraper plugins must use `requests` with HTTPS endpoints, if supported by the external service.

8.3 Local File Permissions

- The system logs and generated '.bib' files might contain partial personal data (authors, etc.). The user is responsible for local file permission settings.

9 Performance Considerations

9.1 Profiling Strategy

- Use Python's `cProfile` or a similar tool to identify bottlenecks when processing large datasets (e.g., 100K references).
- Focus on:
  1. `AbstractAugmenter` concurrency for I/O requests.
  2. `FilterEngine` if classification is CPU-heavy.

9.2 Caching and Bulk Operations

- **Local Cache** for storing `(title, doi) -> abstract`, reducing repeated calls if the pipeline re-runs.
- If external APIs support *bulk queries*, we can design specialized scrapers for batch retrieval. This is an optimization that can be implemented if needed.

9.3 Concurrency Approaches

- Default: `ThreadPoolExecutor` for augmentation.
- CPU-bound tasks for classification: `multiprocessing` or possibly GPU-based inference if the classification model is large.
- The design is flexible; these concurrency details are set in `config` or command-line arguments.

10 Testing and Quality Assurance

10.1 Unit Testing

- Each module has a corresponding `test_MODULE.py`:
  - `test_inputparser.py`: checks whether malformed '.bib' is handled, references are parsed correctly.
  - `test_abstractaugmenter.py`: mocks external HTTP calls to ensure logic for fallback is correct.
  - `test_filterengine.py`: tries references with known titles/abstracts to confirm lexicon or classification thresholds.
  - `test_exporter.py`: checks that the final '.bib' is syntactically valid.

10.2 Integration Testing

- **End-to-End Tests**: Provide a small directory of '.bib' files, run the entire pipeline with real scrapers or mocked scrapers, confirm the final '.bib' matches expected references.
- **Multiple Config Scenarios**:
  - Enable/disable classification.
  - Use different concurrency settings.
  - Different lexicon expansions.

10.3 Code Review and Static Analysis

- `pylint`, `flake8` or `black` for code formatting and basic checks.
- Pull requests require at least one reviewer's approval (best practice in version control).

10.4 Performance Testing

- Large-scale input tests (10K, 50K references) to measure total runtime and memory usage.
- Monitor logs for time spent in augmentation or classification steps.

11 Implementation Plan

11.1 Step-by-Step Guidance

1. **Set Up Repository Structure**:

```
aggregator/
  config/
  modules/
    input_parser.py
    abstract_augmenter.py
    filter_engine.py
    exporter.py
    plugins/
      scrapers/
      classifiers/
    shared/
  tests/
  main.py
  README.md
```

2. **Develop and Test `InputParser`**:
   - Use `bibtexparser` to parse small sample '.bib' files.
   - Implement `parse_references()`, apply invariants.
   - Write unit tests to handle corner cases (empty files, malformed entries).
3. **Implement Basic `AbstractAugmenter`**:
   - Start with a `SemanticScholarScraper` plugin as a template.
   - Validate fallback logic.
   - Use `unittest.mock` or `responses` to simulate external API calls in tests.
4. **Add `FilterEngine`**:
   - Implement `LexiconFilter`.
   - Integrate user-provided lexicons.
   - Write tests verifying that references with known keywords are accepted.
5. **(Optional) `ClassifierFilter`**:
   - Either train a small model or mock the `predict_relevance()` for testing.
   - Integrate threshold logic in `FilterEngine`.
6. **Implement `BibExporter`**:
   - Convert final references back to '.bib' format.
   - Unit test with a handful of references.
7. **Finalize CLI / `main.py`**:
   - Accept `--config` or environment-based configs.
   - Orchestrate the pipeline steps: parse -> augment -> filter -> export.
8. **Integration & Performance Testing**:
   - Use a sample dataset of thousands of references.
   - Measure performance. Optimize if needed (concurrency, caching).

11.2 Version Control Guidelines

- Use **Git** for version control.
- Enforce **feature branches** for each module or functionality.
- Require **Pull Requests** with code reviews to merge changes into `main` or `master`.
- **Continuous Integration** with a service like GitHub Actions or GitLab CI to run tests automatically on each push.

12 Appendices

12.1 Potential Data Structures for Multilingual Support (Future Work)

While the current design mainly assumes *English text*, future expansions might store language codes or use multilingual text classification models. The pipeline design can be adapted by:

- Tagging references with a language field (e.g., `lang = "en"`).
- Storing separate lexicons per language.
- Using a multi-language classification model if references come from non-English sources.

12.2 Sample Config File

```
root_dir: "./references"
output_file: "curated.bib"
invariants:
  require_title: True
  require_year: False

lexicon_paths:
  - "./lexicons/physics.txt"
  - "./lexicons/ml.txt"

concurrency: "thread"
classifier_enabled: true
classifier_threshold: 0.7

# External service creds
semantic_scholar_api_key: "YOUR_KEY"
openalex_api_key: "YOUR_KEY"
```

12.3 References to Other Docs

- *SRS*: Full requirement definitions (FR-1 through FR-8, plus NFR).
- *SAS*: High-level architecture, concurrency, data flow diagrams, plugin approach.

Conclusion

This SDS specifies the detailed design for a **Python-based** pipeline to parse and filter bibliographic entries at the intersection of *physics and machine learning*. The design emphasizes **modularity**, **pipeline structure**, and **extensibility** via plugins. Coupled with robust error handling, logging, and concurrency options, the system can be scaled for large corpora while maintaining minimal false negatives and providing a curated, augmented ‘.bib‘ output for end users.