Connor Boulais
CSC-305
Keen

Final Analysis

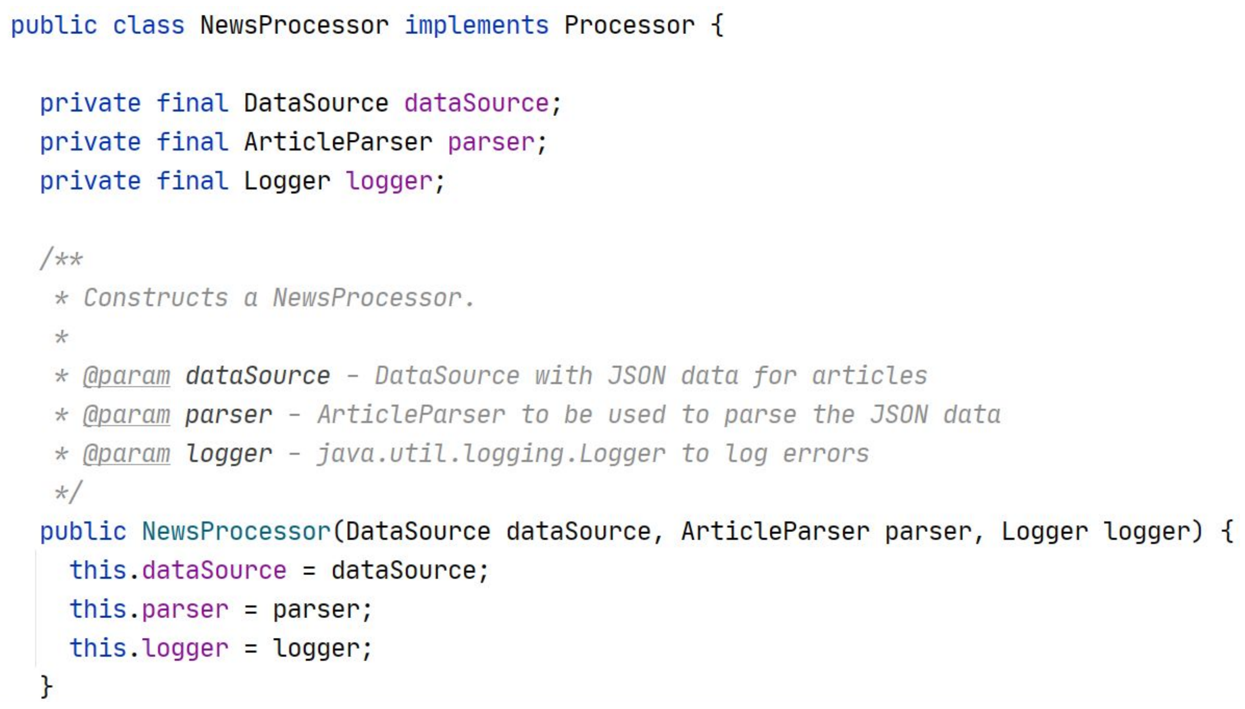
Introduction

This analysis will examine the quarter-long online news aggregator project for CSC-305. Specifically, it will identify the design principles and patterns used, what problems they solve, their benefits, and offer a comparison with possible alternatives.

Decorator

Problem: In this project, we want to perform different types of operations on the news articles we receive, including filtering based on a given filter expression and caching recently seen articles to be removed from subsequent calls. To address this problem, I used an instance of the Decorator pattern by creating multiple processor classes, with each class performing a unique operation, that can be combined into a single processor object that is capable of executing a sequence of operations.

Code:

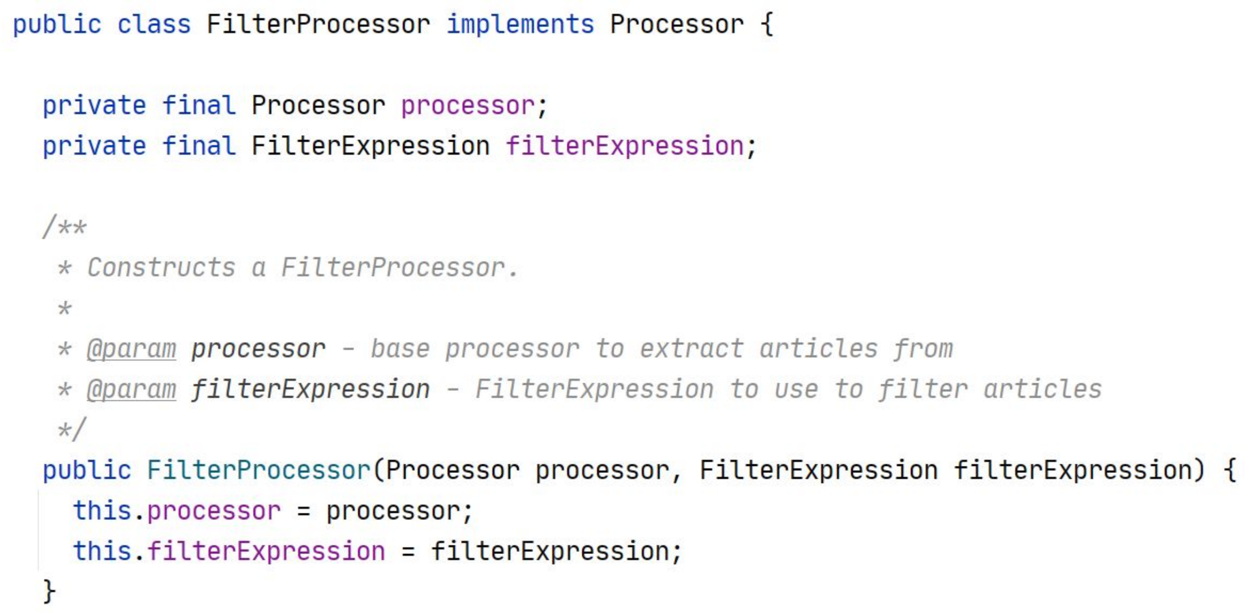


```java
public class NewsProcessor implements Processor {

  private final DataSource dataSource;
  private final ArticleParser parser;
  private final Logger logger;

  /**
   * Constructs a NewsProcessor.
   *
   * @param dataSource - DataSource with JSON data for articles
   * @param parser - ArticleParser to be used to parse the JSON data
   * @param logger - java.util.logging.Logger to log errors
   */
  public NewsProcessor(DataSource dataSource, ArticleParser parser, Logger logger) {
    this.dataSource = dataSource;
    this.parser = parser;
    this.logger = logger;
  }
```

Above is the base NewsProcessor class, which will live at the end of the Decorator "line" of objects. This class simply extracts articles from a DataSource object, using the given ArticleParser.

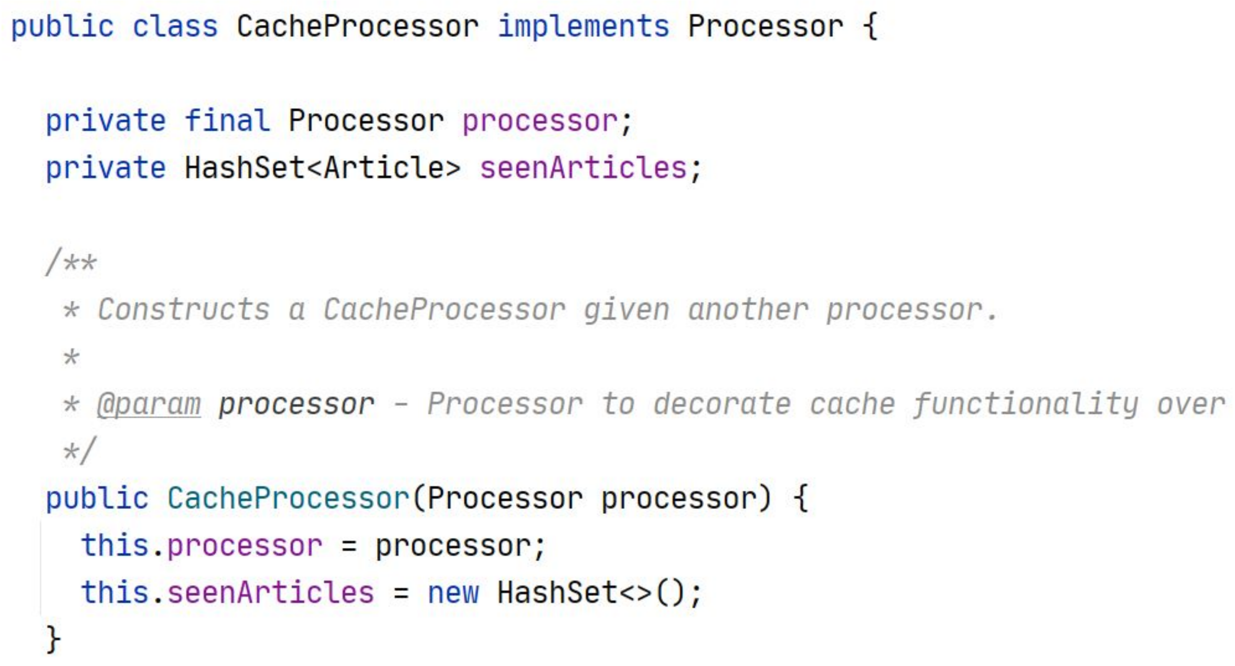


```java
public class FilterProcessor implements Processor {

  private final Processor processor;
  private final FilterExpression filterExpression;

  /**
   * Constructs a FilterProcessor.
   *
   * @param processor - base processor to extract articles from
   * @param filterExpression - FilterExpression to use to filter articles
   */
  public FilterProcessor(Processor processor, FilterExpression filterExpression) {
    this.processor = processor;
    this.filterExpression = filterExpression;
  }
```

Here is the FilterProcessor, which is the first instance of a Decorator. It takes in another processor, to initially extract articles from, but the class will then filter out articles using the given FilterExpression and return the resulting list of filtered articles.



```java
public class CacheProcessor implements Processor {

  private final Processor processor;
  private HashSet<Article> seenArticles;

  /**
   * Constructs a CacheProcessor given another processor.
   *
   * @param processor - Processor to decorate cache functionality over
   */
  public CacheProcessor(Processor processor) {
    this.processor = processor;
    this.seenArticles = new HashSet<>();
  }
```

The last Decorator class is the CacheProcessor, which will extract articles from the given processor while also filtering out recently seen articles.

Benefits of Decorator:

- Allows for the combining of different operations into a single processor
- Allows the program to determine the behavior of processors at runtime, by using different sequences of decorators
- Maintains the Single Responsibility principle by providing separate classes for different processor operations
- Maintains the Open-Closed principle by allowing the extension of processor functionality by adding new processor decorators, without needing to modify existing processor classes.

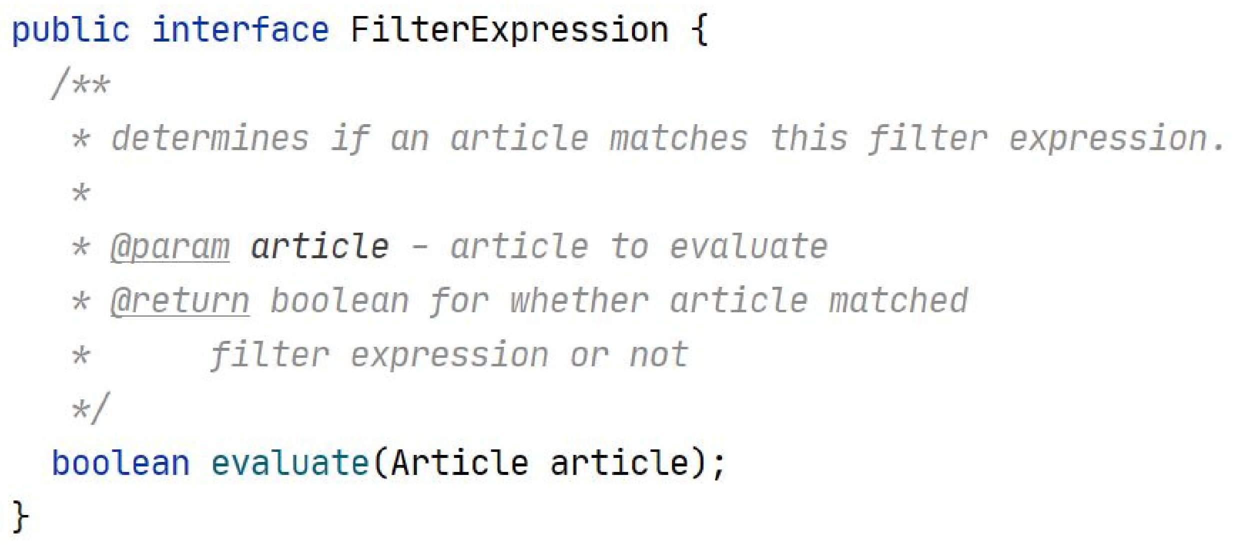
Alternatives to Decorator: Instead of using the Decorator pattern, an alternative would be to create a single NewsProcessor class that performs data extraction, article filtering, and article caching all together (Note: for articles that do not have a filter expression, an EmptyFilterExpression object would be passed to the processor). This alternative would make instantiation of the processor simple, only needing a single call to the constructor of the only NewsProcessor class, instead calling multiple constructors to decorator multiple processors over each other.

However, this approach would have a number of drawbacks. Having a single NewsProcessor class as described means there is no way to only use a subset of decorated operations, or to determine the processor functionality at runtime, as the single call to extract will always filter and cache. Additionally, adding new operations would involve changing the single NewsProcessor class, which would violate the Open-Closed principle and could introduce a bug.

Composite

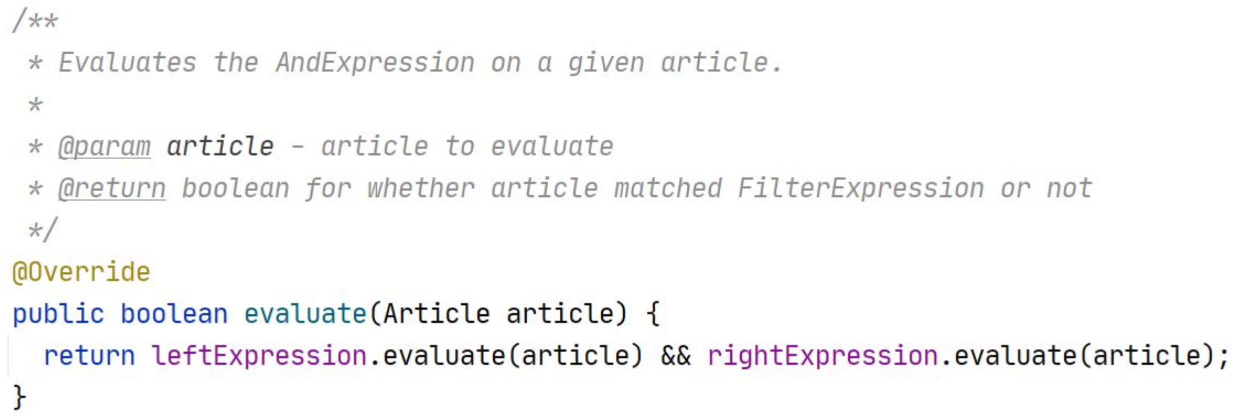
Problem: As mentioned before, one of our processors is a FilterProcessor, which we want to filter out articles based on a given filter expression. This filter expression includes keywords, AND and OR operators, and parentheses to string together more complex expressions. We should be able to pass in a representation of this filter expression and a list of articles to the FilterProcessor, and get back a filtered list of articles. To achieve this, I utilized the Composite design pattern to create a tree of FilterExpression nodes that can filter a list of articles.

Code:



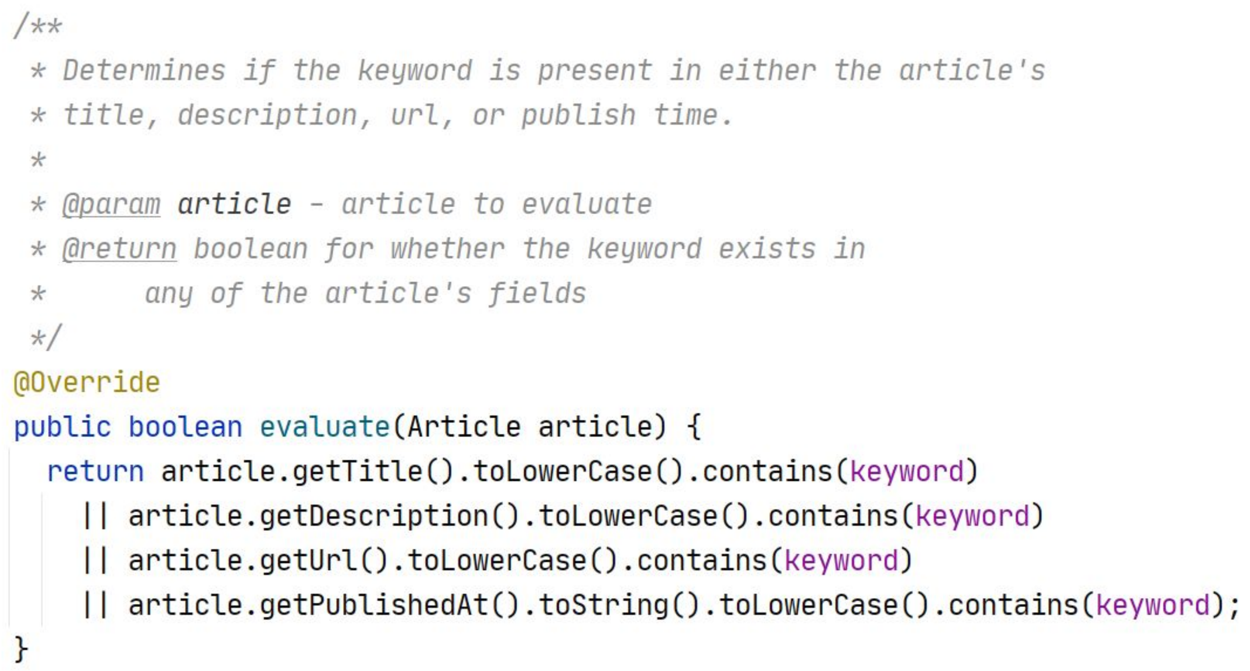
```java
public interface FilterExpression {
  /**
   * determines if an article matches this filter expression.
   *
   * @param article - article to evaluate
   * @return boolean for whether article matched
   *      filter expression or not
   */
  boolean evaluate(Article article);
}
```

Here is the FilterExpression interface, which each node in the tree will implement. It has a single method, evaluate() that takes in an article and returns true if that article matches the FilterExpression, and therefore needs to be filtered out.



```java
/**
 * Evaluates the AndExpression on a given article.
 *
 * @param article - article to evaluate
 * @return boolean for whether article matched FilterExpression or not
 */
@Override
public boolean evaluate(Article article) {
  return leftExpression.evaluate(article) && rightExpression.evaluate(article);
}
```

Above is the implemented evaluate method in the AndExpression class. Here we can see how each interior node in the FilterExpression tree will propagate an evaluate() call to its children. For nested expressions like AND, the result of evaluate() is simply the result of applying the boolean && to the evaluation of the left and right operands of the AND expression.



```java
/**
 * Determines if the keyword is present in either the article's
 * title, description, url, or publish time.
 *
 * @param article - article to evaluate
 * @return boolean for whether the keyword exists in
 *      any of the article's fields
 */
@Override
public boolean evaluate(Article article) {
  return article.getTitle().toLowerCase().contains(keyword)
    || article.getDescription().toLowerCase().contains(keyword)
    || article.getUrl().toLowerCase().contains(keyword)
    || article.getPublishedAt().toString().toLowerCase().contains(keyword);
}
```

Here is the evaluate method for the KeywordExpression class, which will sit at the leaves of the FilterExpression tree. We can see this evaluate method does not pass on the request to any children (since it will be a leaf node) and handles the request by determining if the given keyword exists in any of the article's attributes.

Benefits of Composite:

- Does not require any complex loop to filter articles, as the FilterExpression objects themselves cover the whole tree by passing filter requests down to their children.
- Maintains the Open-Closed principle by allowing for the addition of new filtering mechanisms without changing the existing filter operations. Simply create a new class that implements FilterExpression and add it to the tree.

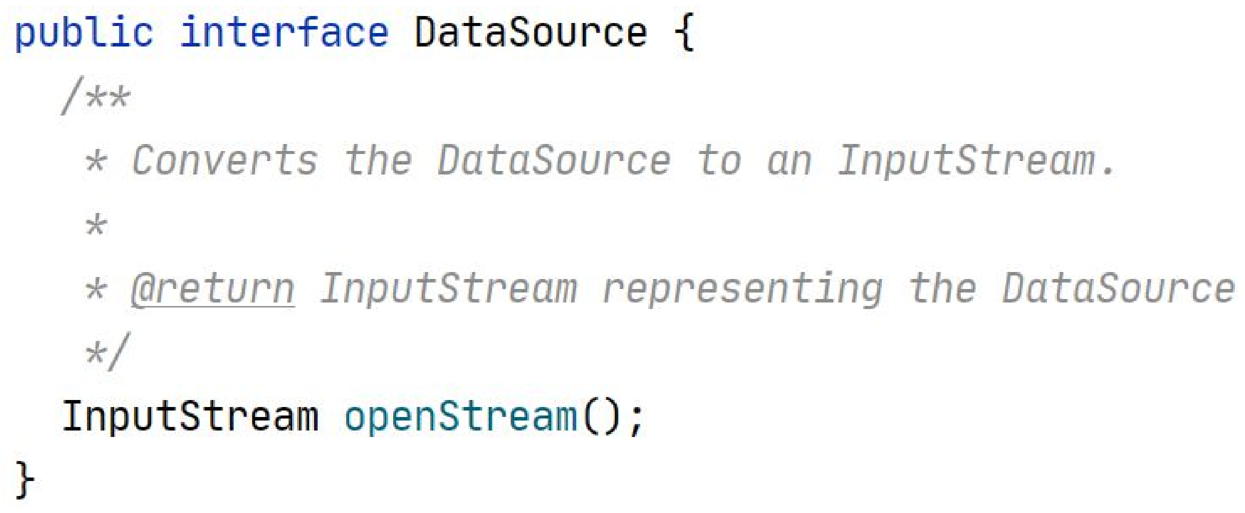
Alternatives to Composite: Instead of creating a set of FilterExpression classes for a composite tree, we could have written a single method (recursive or iterative) that takes a filter string directly from the configuration file and a collection of articles, and filters the list according to the filter string. The method would likely have a sequence of conditionals that handle every possible operand and operator it could come across.

The benefit of this approach is that there is no complex tree of objects (or any of the files that go along with the FilterExpression tree), just a single filter method that takes a string and collection of articles. Therefore, in regards to relationships between classes in the project, complexity is reduced. Additionally, the single method approach means we do not have to construct the FilterExpression tree from the configuration file, as we only need to grab the filter expression string directly from the file and pass it to our single filter method.

However, this approach also has a number of downsides. For one, using this single method would involve a complex recursive or iterative algorithm that would be more difficult to develop and maintain than any of the logic in the FilterExpression classes alone. Additionally, if a large number of filter operations are used, then the single filter method would contain an awkwardly long string of conditionals that is difficult to maintain. Lastly, if we decide in the future to add more types of filter operations, we would have to modify the single filter method, which violates the Open-Closed principle and could introduce a bug.

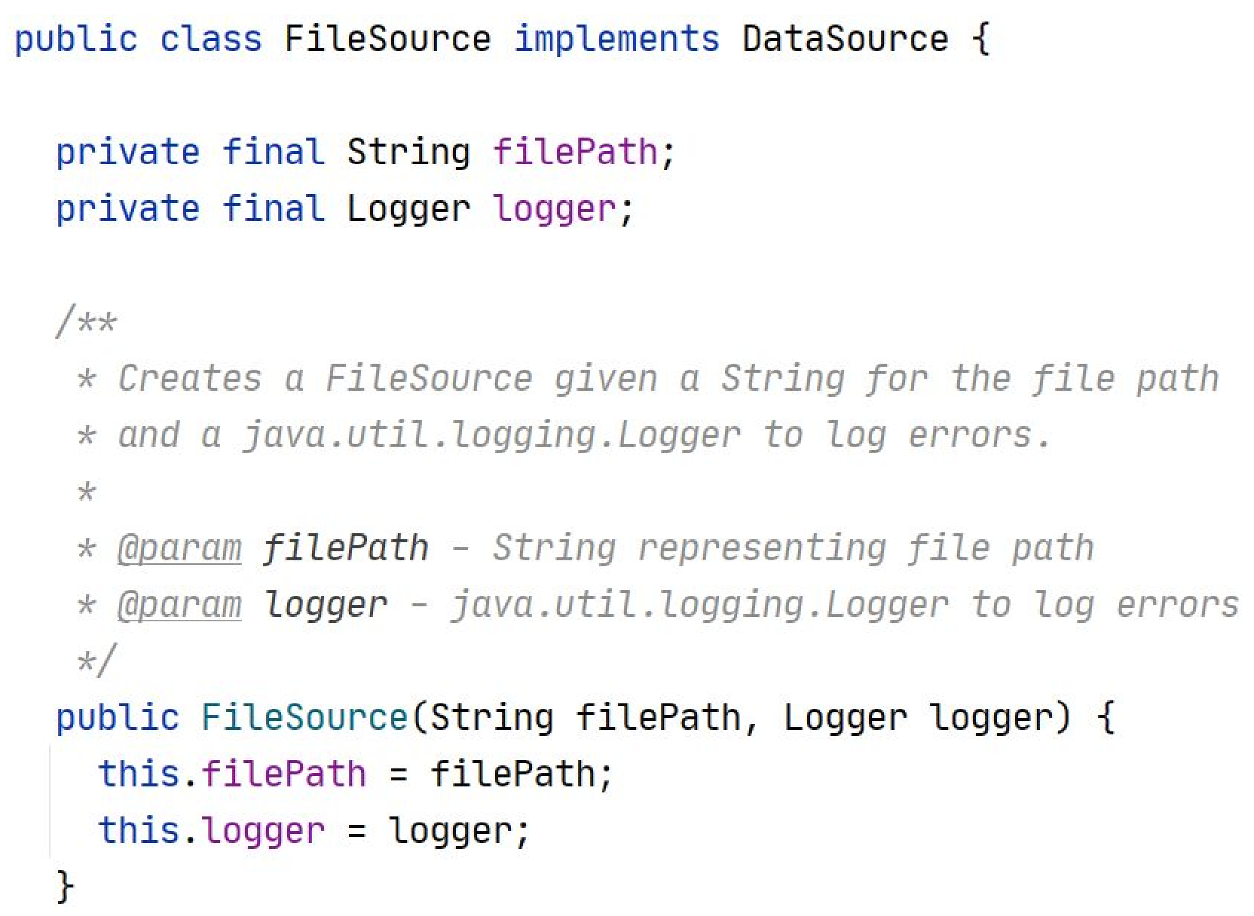
Dependency Inversion
Problem: The NewsProcessor in this project extracts a list of articles from a source of JSON data, however we want our NewsProcessor to be able to read this JSON from different types of sources, including files and URLs. To do this, I employed the Dependency Inversion principle of the S.O.L.I.D. framework to create a DataSource class that decouples the NewsProcessor class from any specific type of JSON source. As a result, the NewsProcessor depends only on the abstracted DataSource interface, rather than any concrete source type.

Code:



```
public interface DataSource {
  /**
   * Converts the DataSource to an InputStream.
   *
   * @return InputStream representing the DataSource
   */
  InputStream openStream();
}
```

Above is the DataSource interface, which contains only a single method to open an InputStream on the data source.

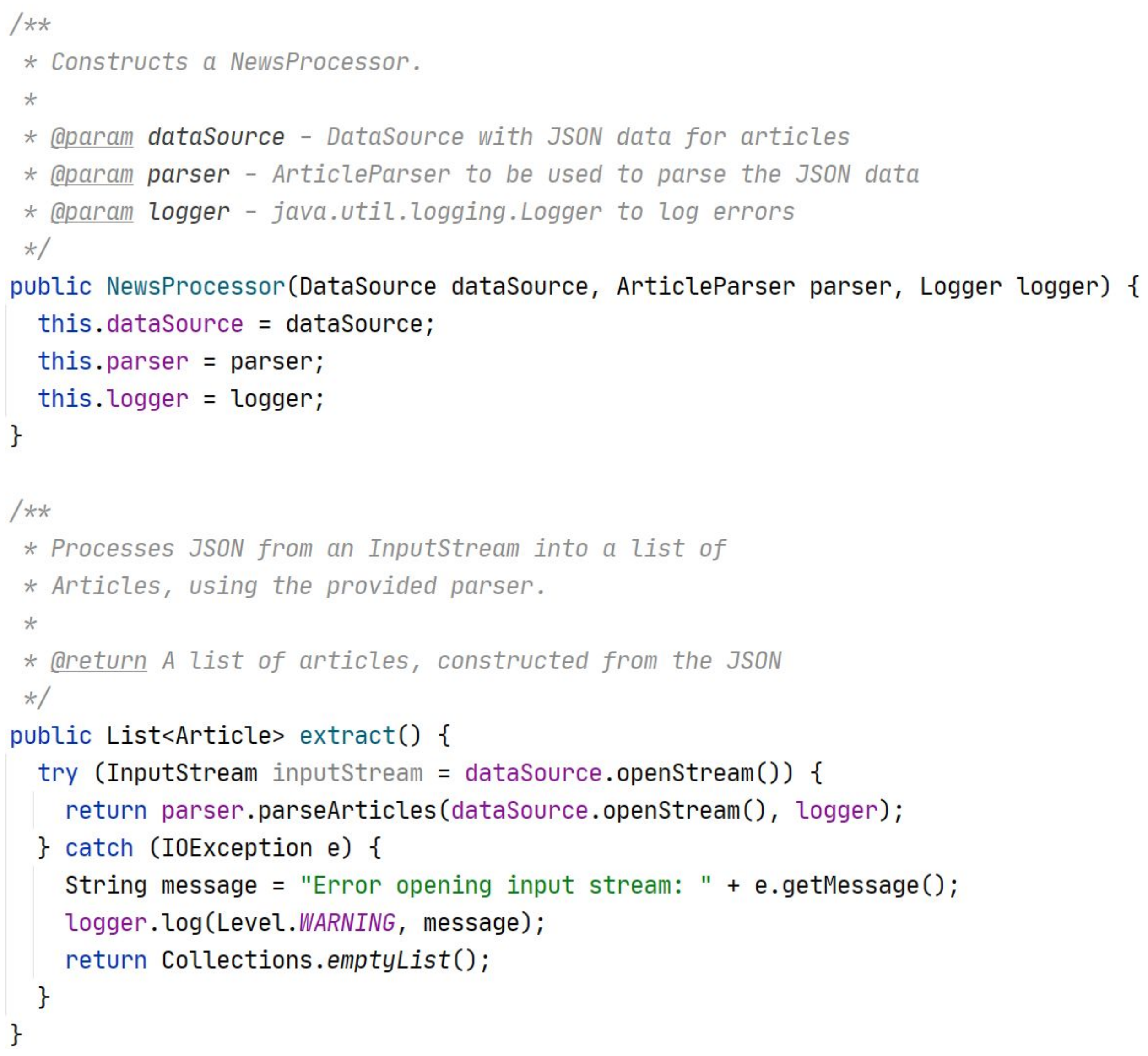


```java
public class FileSource implements DataSource {

  private final String filePath;
  private final Logger logger;

  /**
   * Creates a FileSource given a String for the file path
   * and a java.util.logging.Logger to log errors.
   *
   * @param filePath - String representing file path
   * @param logger - java.util.logging.Logger to log errors
   */
  public FileSource(String filePath, Logger logger) {
    this.filePath = filePath;
    this.logger = logger;
  }
```

Here we can see an instance of the DataSource interface, the FileSource class, which is an abstraction of a file.



```java
/**
 * Constructs a NewsProcessor.
 *
 * @param dataSource - DataSource with JSON data for articles
 * @param parser - ArticleParser to be used to parse the JSON data
 * @param logger - java.util.logging.Logger to log errors
 */
public NewsProcessor(DataSource dataSource, ArticleParser parser, Logger logger) {
  this.dataSource = dataSource;
  this.parser = parser;
  this.logger = logger;
}

/**
 * Processes JSON from an InputStream into a list of
 * Articles, using the provided parser.
 *
 * @return A list of articles, constructed from the JSON
 */
public List<Article> extract() {
  try (InputStream inputStream = dataSource.openStream()) {
    return parser.parseArticles(dataSource.openStream(), logger);
  } catch (IOException e) {
    String message = "Error opening input stream: " + e.getMessage();
    logger.log(Level.WARNING, message);
    return Collections.emptyList();
  }
}
```

Above we can see how the NewsProcessor class depends only on the abstracted DataSource interface, rather than any specific type of source. A DataSource is passed in to the NewsProcessor constructor at the top, and then is used in the extract() method to obtain an InputStream from the source.

Benefits of Dependency Inversion:

- Allows for adding new types of data sources without changing the NewsProcessor class, maintaining the Open-Closed principle
- Allows for testing the NewsProcessor class without any external dependencies, because a mocked data source can be passed to the NewsProcessor instead of a valid file or URL.

Alternatives to Dependency Inversion: An alternative to creating the DataSource class would be to implement an overloaded constructor in the NewsProcessor class, one that takes a File object and another that takes a URL. Both of these constructors would open an InputStream from the given source and store the InputStream as an attribute in the object. Then the extract() method would use the object's stored InputStream as a JSON source.

The benefits of this approach would be not having to define an extra interface abstraction, therefore simplifying the class relationships in the project. Additionally, using this overloaded constructor would arguably make construction of the NewsProcessor simpler for the client, as they only need to pass in a common Java File or URL which they are likely familiar with, rather than a newly defined DataSource interface.

There are many downsides to this approach though. For one, the InputStream is opened early, and could remain open for a while before it is used. Moreover, the NewsProcessor class would need logic to open an InputStream from File and URL objects, and logic to handle when those objects are invalid, which potentially violates the Single Responsibility principle. The most significant drawback though, is that adding new types of DataSources means changing code in the NewsProcessor class which violates the Open-Closed principle and could introduce a bug.

Conclusion

The news aggregator project demonstrates a number of different design patterns and princples, and this document has outlined their benefits as well as possible alternatives. Overall, the benefits of the Decorator pattern, Composite pattern, and Dependency Inversion principle outweighed each of their respective alternatives, and therefore were the best choice for this project.