

RE-ENGINEERING LAB

Reverse Engineering Apache Lucene

CURRENT TOPICS IN SOFTWARE ENGINEERING
SOFTWARE EVOLUTION

SS 2014



28. April 2014

1 Initial Understanding

To get a first impression of the project, we answer the following questions.

1.1 Main Features of Apache Lucene

Apache Lucene consists of several modules (core, solr, pyLucene, etc), but we are concentrating here only on *Apache Lucene Core*. To quote the webiste:

"Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform."

So with Apache Lucene you can build an index over large scale unstructured text corpora and perform various kinds of queries on it.

It claims to be very conservative concerning resource allocations. The major advantages are that it is full cross-platform compatible and the fact that it is open source software, licensed under the Apache License, which allows full commercial and open-source use.

1.2 Read the Code in one Hour

We stepped through all the packages in Eclipse using Reference Browsing, Outlining and Type hierarchy features. Especially we were very pleased about the huge amount and well structured test suites we found. We clicked on the very big visualized classes by X-Ray and could then follow the code in the source editor, so we already saw our big enemies in the on hour reading pattern.

Almost every method was documented not just using jAutoDoc but rather contained useful explanations of what the parameters and the method in general does.

1.3 Do a Mock Installation

Right after we checked out the source code from SVN trunk we tried to build the system.

The build system is based on *Apache ANT* for building and *Apache IVY* for dependency and package managing. It is a nice feature that even though Apache ivy is not installed on the machine you can retrieve and install it right away using a provided ant task *ivy – bootstrap*

After that ivy loaded all dependencies for lucene and ANT mastered the building process without any complications.

In the next step we generated the eclipse project to start browsing the code. During that we started the junit tests to see if everything works as it should.

So basically installing and running the system was very easy and without any major trouble.

1.4 Which are the important source code entities?

By looking at the most primitive example provided at the root of the lucene documentation, you already get a feeling for the most important entities:

- *Directory* is an abstract class providing the storage wrapper for the built up index
- *Index-Writer* is the one actually responsible for writing the index.
- *Query* is an abstract class for encapsulating what the user is searching for.
- *Document* a query is performed over several documents.

This already gives us a good overview of what the key concepts of apache lucene are.

We started getting a first impression by loading the source code into *X-Ray* and *SonarQube*. This already gave us an first impression about the size and complexity of the project.

1.5.1 Quality of the Implementation

As you can see in the screenshot, SonarQube qualified lots of the core modules with green (meaning good quality).

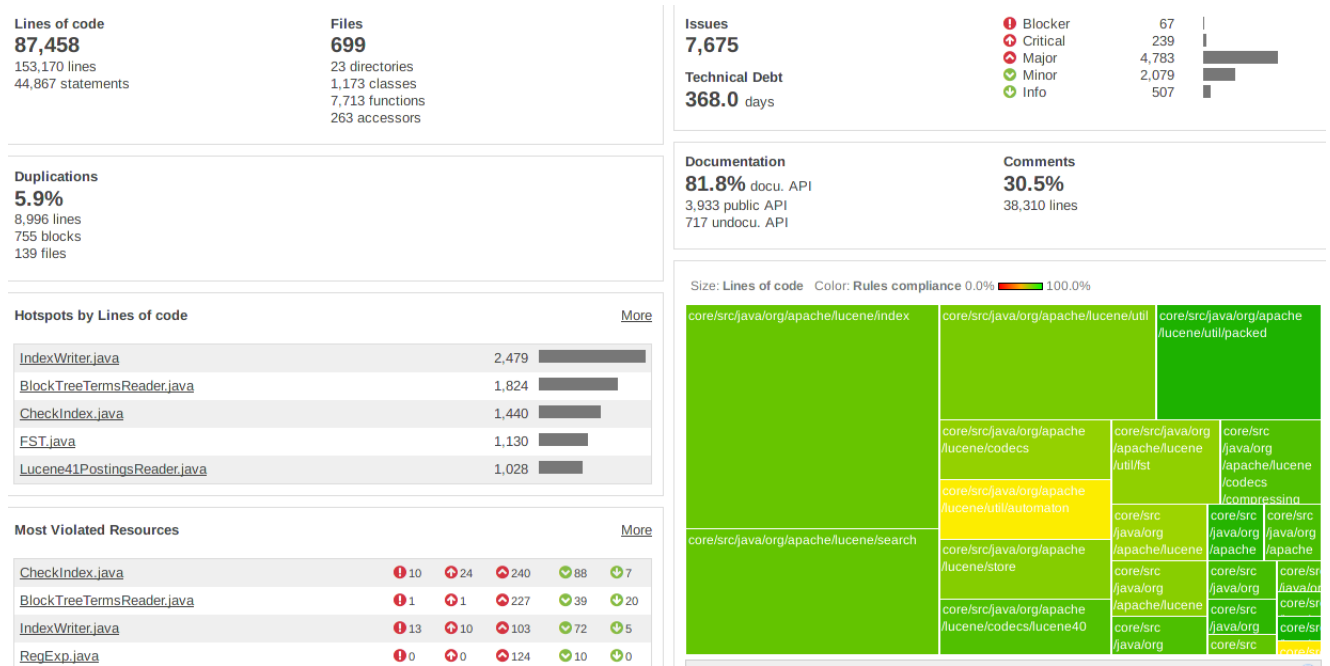


Figure 1: Screenshot taken from the Dashboard of SonarQube

The core metrics showed us, that the implementation is in quite good shape, but there are also some exceptions.

- There is lots of documentation (30.5 % of all source lines are comments), which was not auto-generated, but has meaningful descriptions for each method and class.
- The average McCabe complexity per function is 2.8, which is very low

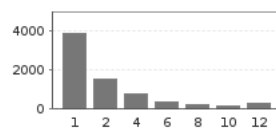


Figure 2: Most of the methods have a Complexity below 3

- The rule violations showed by SonarQube are quite low for most of the classes.
- There are lots of test suites and cases with very good coverage at about 84 % in total. We checked for the missing coverage percentage. The missing coverage comes not from untested classes but from unforced try / catch blocks, which is good. So exception behavior of the codes mostly

untested, but without exceptions, the code has a coverage of almost 100 %. In total there are more than 9,500 test cases. Only some of them failed so source code and test cases seem to be quite up to date and synchronized.

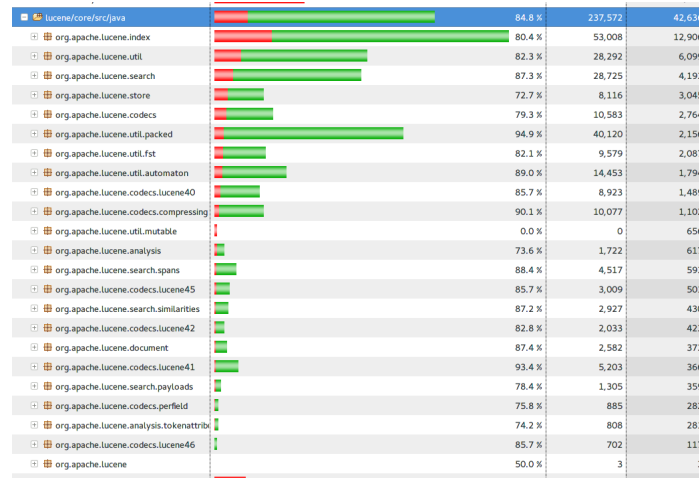


Figure 3: Coverage report using EclEmma

- There seem to be lots of inner classes, because there are only 699 source files but 1,173 classes.
- There are some files which have above 2,500 lines of code.

So in total the quality of the implementation seems to be quite good, except for some files which have lots of lines of code.

1.5.2 Quality of the Design

The design and architecture on the other hand showed major flaws. The previously discovered classes with several thousand lines of code were a first indicator.

The package dependency graph provided by X-Ray gave a second clue that there is really a mess in the design of this project.

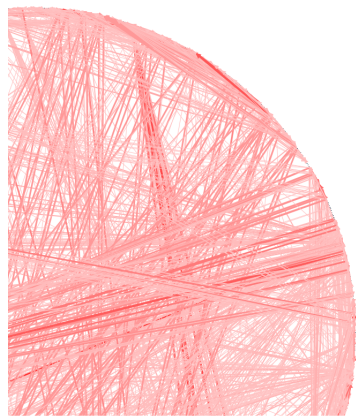


Figure 4: Screenshot from the class dependency graph in X-Ray - Everything needs everything

When we ran further analysis using *inCode* which supported our hypothesis. It detected lots of design flaws like god classes, feature envy, duplications, etc.

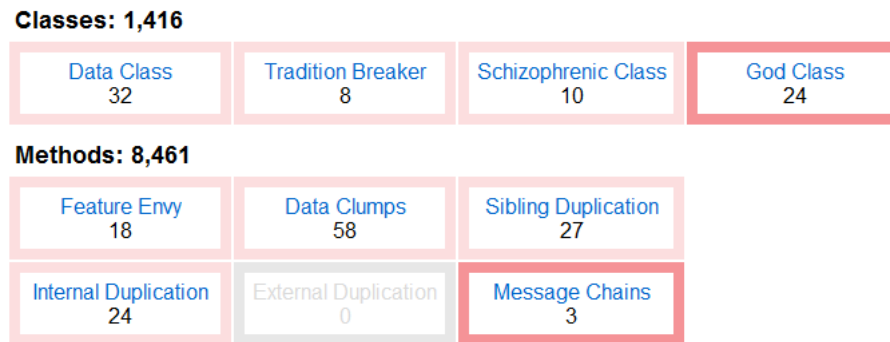


Figure 5: Screenshot of the project overview from the inCode analysis

So the architecture really is very bad.

1.6 Is a re-engineering feasible?

Well to answer the more simple question: *Is reengineering required?* It is a clear *Yes!*. A rewrite is off the table, lucene is not just a project that is rarely used.

It can definitely not be done at once, there are so many problems. We think lots of improvements can already be done by doing local refactoring, splitting up god classes and long methods into their own pieces.

The very good code coverage of the test cases definitely supports us doing a re-engineering, so we can be sure we don't make things worse.

1.7 What are the exceptional packages, classes and methods?

There are some classes which are very exceptional, because they have a very high number of lines of code.

- org.apache.lucene.index.IndexWriter: 2,479 LOC
- org.apache.lucene.codecs.BlockTreeTermsReader: 1,824 LOC (lots of them duplicated)
- org.apache.lucene.index.CheckIndex: 1,440 LOC

1.8 What does the inheritance structure look like?

InCode provides a very nice feature called *Inheritance Map* which gives you a very good first impression of the inheritance structure.

As illustrated, the hierarchies are not very deep (3 or 4) but there are lots of classes that are not part of any inheritance hierarchy. The red ones illustrate classes with many or very critical design flaws.

One of the deepest hierarchies are provided for the different type of lucene queries. This seems to be very reasonable. Others represent different types for Iterators over various lucene-internal data structures.

1.9 Tools

- X-Ray: Used for the first contact with the project. It does not provide a good overview.
- SonarQube: Used to get a first impression of the size of the project and the source code quality.

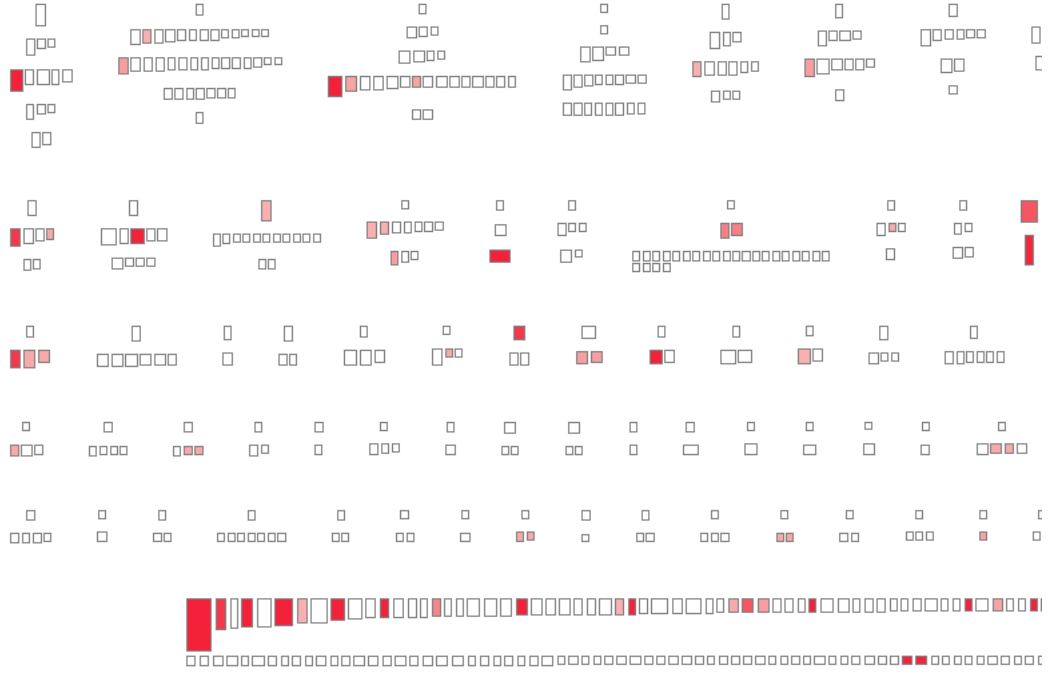


Figure 6: Inheritance Map of Apache Lucene

- inCode: Used to get a better overview over class hierarchies, packages and so on.
- inFusion: Tried, but project was too big for the evaluation version.
- CodeCity: How we understood it, we need inFusion to build the model of the project which can then be displayed in CodeCity.

2 Detailed Model Capture

- How three main features of the subject system are implemented (mention the main classes, methods and briefly describe how they depend on each other).

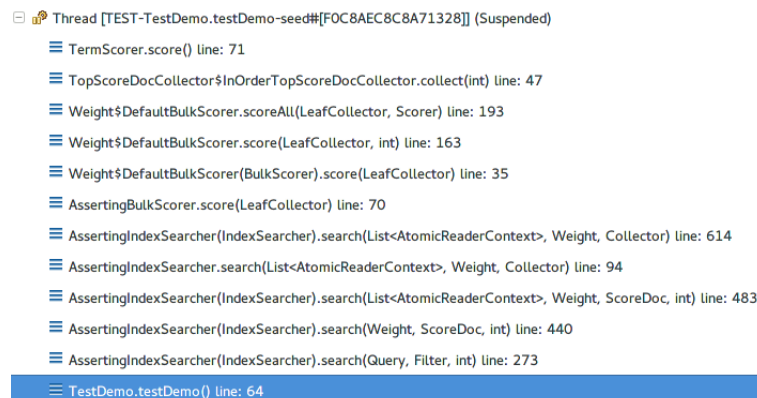
2.1 Step through the Execution

For the step-through execution we used the *org.apache.lucene.TestDemo* test case and set break points. We wanted to know how the indexing is performed and how the later searching based on the index is performed.

Therefore this was the perfect entry point.

From this we learned, that the index writer is transaction based, and that concurrent modifications to it are synchronized.

Also each found document gets rated by its relevance it has for the performed query. Those are returned in a class called *TopDocs*. For calculating the relevance, several *Scorer* classes are available.



```
Thread [TEST-TestDemo.testDemo-seed#[FOC8AEC8C8A71328]] (Suspended)
  TermScorer.score() line: 71
  TopScoreDocCollector$InOrderTopScoreDocCollector.collect(int) line: 47
  Weight$DefaultBulkScorer.scoreAll(LeafCollector, Scorer) line: 193
  Weight$DefaultBulkScorer.score(LeafCollector, int) line: 163
  Weight$DefaultBulkScorer(BulkScorer).score(LeafCollector) line: 35
  AssertingBulkScorer.score(LeafCollector) line: 70
  AssertingIndexSearcher(IndexSearcher).search(List<AtomicReaderContext>, Weight, Collector) line: 614
  AssertingIndexSearcher.search(List<AtomicReaderContext>, Weight, Collector) line: 94
  AssertingIndexSearcher(IndexSearcher).search(List<AtomicReaderContext>, Weight, ScoreDoc, int) line: 483
  AssertingIndexSearcher(IndexSearcher).search(Weight, ScoreDoc, int) line: 440
  AssertingIndexSearcher(IndexSearcher).search(Query, Filter, int) line: 273
  TestDemo.testDemo() line: 64
```

Figure 7: Stacktrace for a simple search query

By displaying the stacktrace you also get a very good impression of how the call hierarchy is for a simple search example.

2.2 Feature - Indexing

Indexing is one of the core features of lucene. It is the premise for fast searching.

An apache lucene index is built up based on a number of documents. Lucene is very flexible on the serialization target (*org.apache.lucene.store.Directory*). The index can either be build up in memory (*org.apache.lucene.store.RAMDirectory*) or on hard disk (*org.apache.lucene.store.CompoundFileDirectory*). This would theoretically also allow to provide other storage mechanisms just by extending the abstract *org.apache.lucene.store.Directory* class.

Such a directory will be passed to an *org.apache.lucene.index.IndexWriter* which actually does the serialization of the Index to be build up.

This can be done concurrently. The *IndexWriter* is synchronized using a *two-phase commit protocol* to ensure data integrity.

It is important to call the *close()* method of *IndexWriter*, which than tries to merge all added documents and commit cleanly so the index is consistent.

A document (which is added to the directory) can have several different fields, which are all implemented in different subclasses of *org.apache.lucene.document.Field* (e.g. *LongField*, *IntField*, *TextField*).

How the documents are analyzed for the Index is implemented in various subclasses of the *org.apache.lucene.analysis.Analyzer* classes (e.g. *GermanAnalyzer*, *EnglishAnalyzer*, etc.).

2.3 Feature - Searching and Query Formulation

After an index has been built up, lucene can use it to search through large amounts of text using it. There are various ways to search through the documents for content.

The main classes and interfaces used for searching are the following:

- *org.apache.lucene.index.IndexReader*: This abstract class defines the interfaces which are required to read the index. Concrete implementations can either read from the file system or directly from memory (*RAMDirectory*).
- *org.apache.lucene.search.IndexSearcher*: It scans through the index based on the passed on *IndexReader* object.
- *org.apache.lucene.search.Query*: This is most general class for formulating a search. There exist several subclasses which all specialize the search behavior of the *IndexSearcher*. Most important here is the *WildcardQuery*, which can be used in general if we don't know much about the content we are searching through. The type of query influences the searching performance very much.

If the content has little structure, you can organize it in the indexing process through *fields*. Fields provide a good way to encode meta data, which can then also be searched very fast.

2.4 Feature - Configurable Storage Engines (Codecs)

The codecs in lucene provide the possibility to choose between various storage engines used for persisting the index.

When a specific codec is used, it writes its own name into the index, so that for reading the index, the right (same) codec is used.

Basically Lucene brings a new codec for each major version. This ensures, that it is backwards compatible with older built up indexes.

By extending the abstract class *org.apache.lucene.codecs.FilterCodec*, one could also define an own codec for serialization.

Such a codec provides only various getter methods, for formats on each part of the index. Such Formats all have a abstract baseclass which then are extended by each specific codec format subclass.

3 Problem Detection

3.1 SRP violations

The Single Response Principle Violation: A class should have a single purpose and only one reason to change.

We used the Schizophrenic Class View of *inCode* to find problematic classes. *inCode* detected 10 classes that violate the SRP principle because these classes have public methods that are used non-cohesively by client methods. Especially disjoint groups of client classes that use disjoint fragments of the class interface in an exclusive manner are focused here. As you can see in Figure 8 and 9 the affected classes exhibit a strong outgoing coupling intensity.

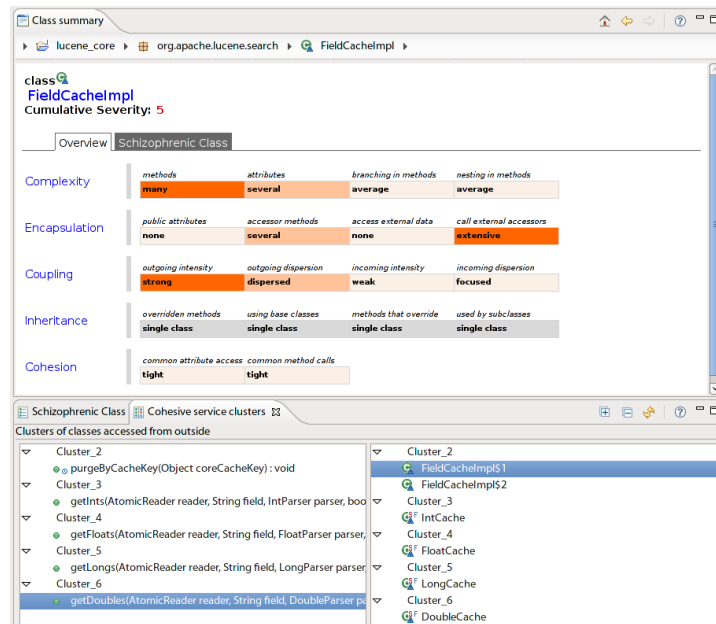


Figure 8: Screenshot of the analysis of class `FieldCacheImpl` in *inCode*

3.2 OCP/LSP violation

The Open-Closed Principle: Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

There are several classes that violate this principle. First of all God Classes are a good starting point to look at. Then we inspected the outgoing and incoming coupling intensities of these classes to find the ones with the highest values (strong). We used the different views provided by *inCode* for this task. Classes that exhibit strong coupling and weak cohesion in addition to referencing and being referenced by many other classes and methods will certainly be problematic as soon as something changes, because there are many interdependencies. `SegmentReader`, `DocumentWriter`, `ReaderAndUpdates`, `DocumentsWriterFlushControl`, `SegmentInfos` and `Automaton` have been identified as violators. See figure 10 for an overview of the most prominent example.

The Liskov Substitution Principle: Subtypes must be substitutable for their base types. If a client uses a base class, then it should not differentiate the base class from derived class.

This principle is violated by the God Class `IndexWriter`. Multiple `instanceof` checks are used in conditions to find the concrete subtype of the super class `IndexReader`. See figure 11 for details on the coupling perspective and see figure 12 for a code extract.

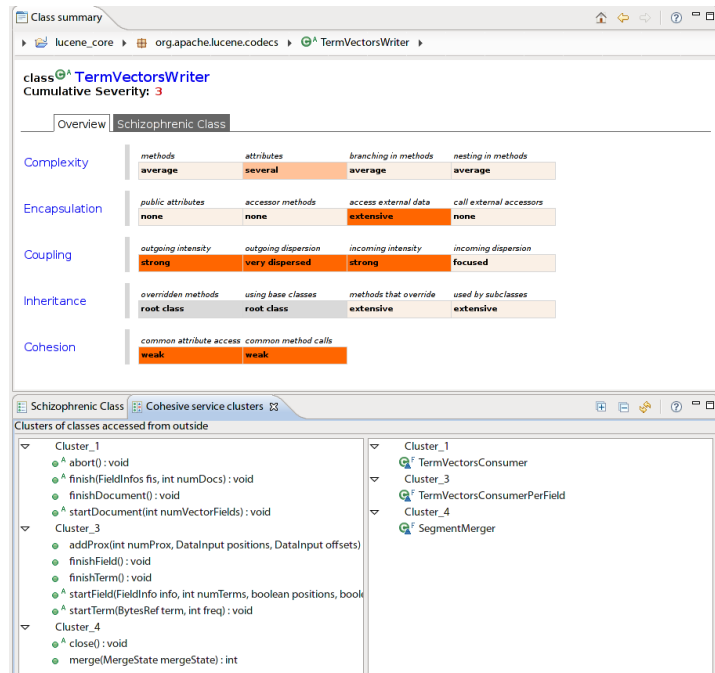


Figure 9: Screenshot of the analysis of class `TermVectorsWriter` in *inCode*

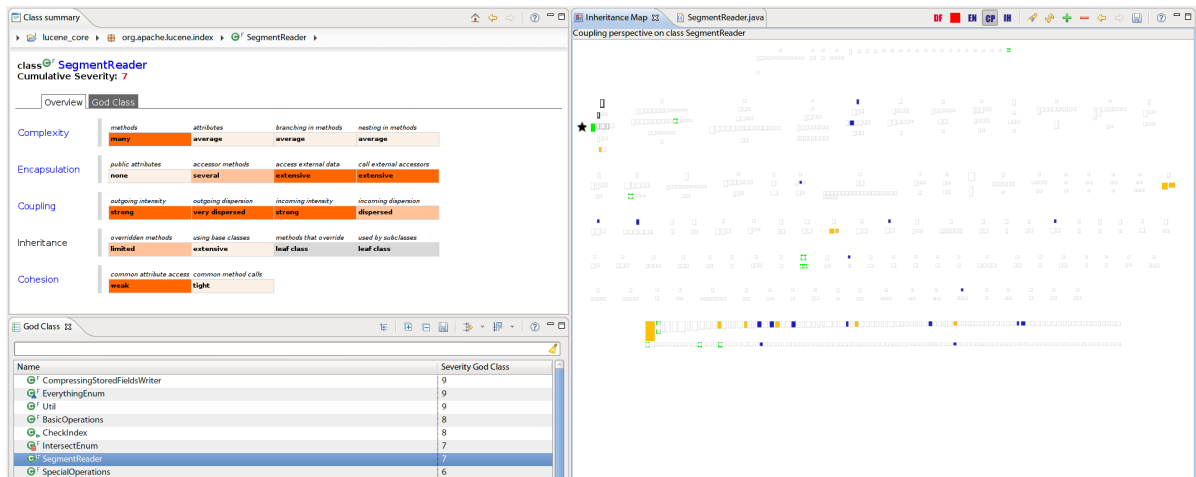


Figure 10: Screenshot of the analysis View of `SegmentReader` in *inCode*

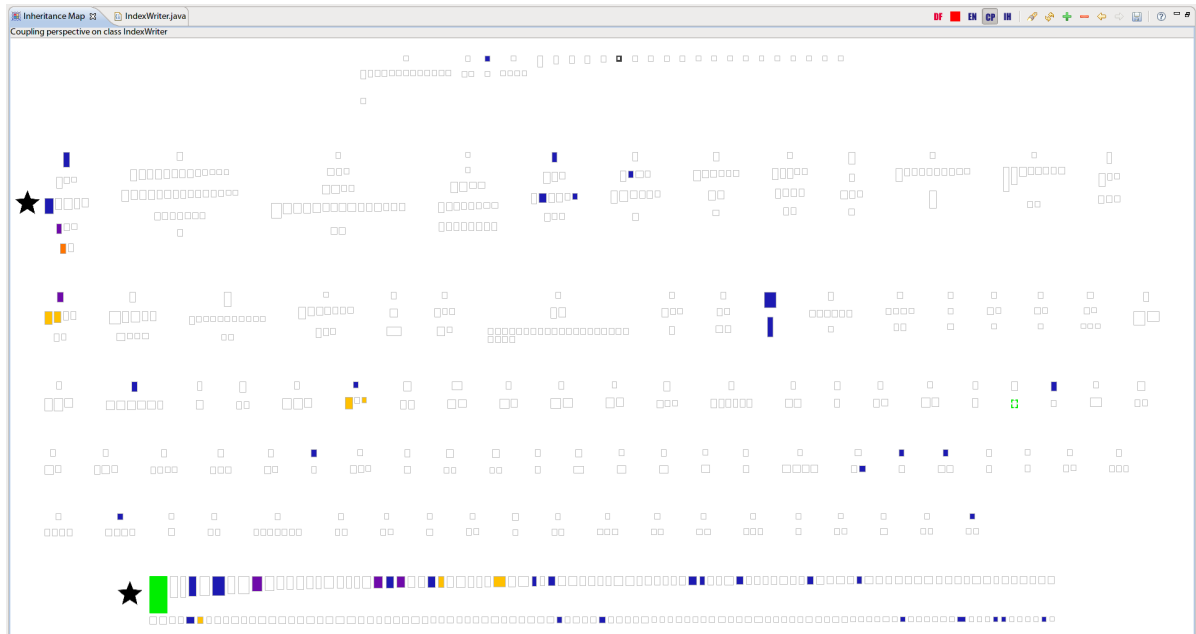


Figure 11: Screenshot of the Coupling View of `IndexWriter` in *inCode*

```
public synchronized boolean tryDeleteDocument(IndexReader readerIn, int docID) throws IOException {
    final AtomicReader reader;
    if (readerIn instanceof AtomicReader) {
        // Reader is already atomic: use the incoming docID:
        reader = (AtomicReader) readerIn;
    } else {
        // Composite reader: lookup sub-reader and re-base docID:
        List<AtomicReaderContext> leaves = readerIn.leaves();
        int subIndex = ReaderUtil.subIndex(docID, leaves);
        reader = leaves.get(subIndex).reader();
        docID -= leaves.get(subIndex).docBase;
        assert docID >= 0;
        assert docID < reader.maxDoc();
    }

    if (!(reader instanceof SegmentReader)) {
        throw new IllegalArgumentException("the reader must be a SegmentReader or composite reader conta");
    }
}
```

Figure 12: Screenshot of the source code extract of `IndexWriter`

3.3 DIP violation

The Dependency Inversion Principle: High-level modules should not depend on low-level modules. Both should depend on abstractions. No variable should hold a reference to a concrete class and no class should derive from a concrete class.

Several classes can be found that violate this principle. We would like to show the example of `BulkOperation` that stores references to an array of its subclasses (`BulkOperationPacked1`, `BulkOperationPacked2`, `BulkOperationPacked3`, ...). These subclasses are all derived from a concrete class called `BulkOperationPacked`, which itself is an instance of `BulkOperation`. See the inheritance diagram shown in figure 13.

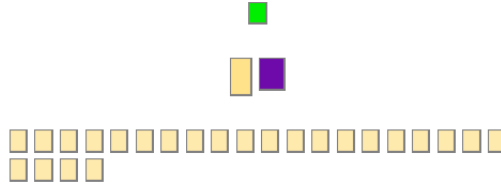


Figure 13: Screenshot of the inheritance perspective of `BulkOperation` in *inCode*

3.4 DRY violation

Don't Repeat Yourself: Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

We used two different approaches to find duplicated code. On the one hand we looked at the Code Duplication View of *SonarQube* and on the other hand we used the Internal Duplication View of *inCode*. Both tools can be used get a deeper insight, but each one has its advantages and drawbacks. Whereas *SonarQube* is focussing on numbers, code fragments and statistics (see figure 14), *inCode* is presenting a more conceptual overview of the duplicated code (see figure 15).

We think that combining the above approaches is a valuable tool for detecting DRY violations. Especially using *SonarQube* to see the overall picture and then using *inCode* to get a more detailed view on these entities particularly with regard to possible future refactoring.

3.5 ISP violation

The Interface Segregation Principle: Clients should not be forced to depend on methods they do not use. The dependency of one class to another one should depend on the smallest possible interface.

`FieldCache` is the interface with the largest number of methods and there is only a single class - `FieldCacheImpl` - that implements this interface. But `FieldCacheImpl` is a Schizophrenic Class, which means that it violates the Single Responsibility Principle. Therefore it would be a good idea to separate concerns and split the interface including the implementation into several different classes to obtain a higher cohesion and less coupling.

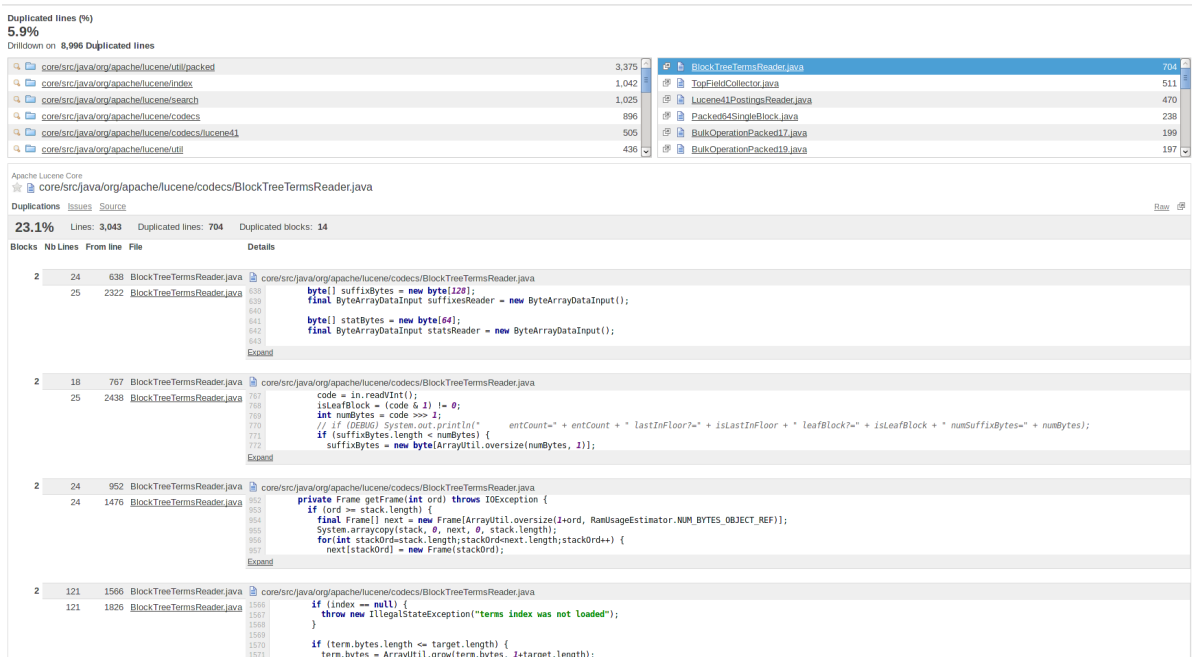


Figure 14: Screenshot of the Code Duplication View of *SonarQube*

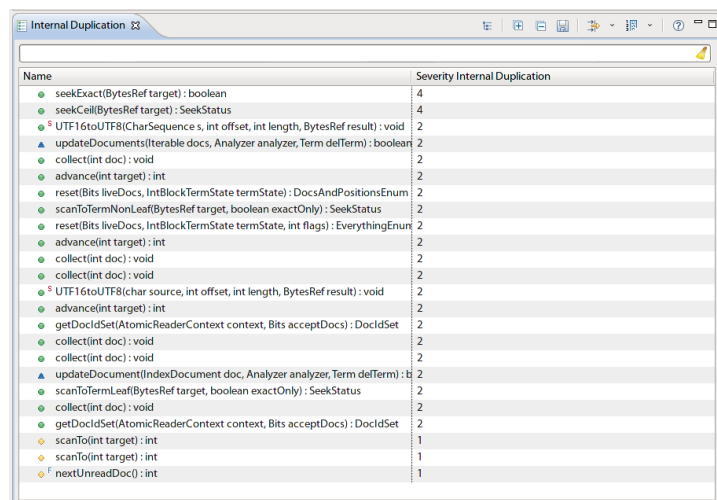


Figure 15: Screenshot of the Internal Duplication View of *inCode*

