Software Maintenance and Evolution
God Components

Jeroen G S Overschie (s2995697)
Konstantina Gkikopouli (s3751309)

January 17, 2021

Repository: `https://github.com/dunnkers/god-components`

Contents

1 Introduction

God Components (or 'God objects') are components in a software system that have accumulated a large bulk of classes and lines of code over time. Such really large, bulky components are hard to maintain and to reason about; they are in fact a software *anti-pattern* [10]. It is preferred to have smaller, isolated components instead. Although it is a common good practice to build software by creating small building blocks of reusable code and accessing them using a declarative and well-documented API, big code-bases might still suffer from scaling issues: large inter-weaved software components might develop that become difficult to reason about.

To prevent God Components in your software, it is at all times important to keep refactoring a code-base at the architecture level, i.e. apply step-wise evolution [6]. Be wary and suspicious about 'vague' abstractions that seem to want to answer too many questions at once [8]. God Components might be broken up into separate, independent components that each have their separate function. This makes the code easier to test and reason about - and above all; more understandable to the humans actually maintaining the code-base.

In this project, one such analysis will be ran on the **Apache Tika** project [1]. Apache Tika is a software package built to detect and extract text and metadata from many different file formats. File formats include PDF, PPT and XLS, and can all be accessed through Tika's API, making it easy to process a large amount of files using just a concise amount of code. Besides its text extraction capabilities, it is also commonly used to classify documents using meta-data obtain from Tika, such as any file's extension [2].

God Components will not only be identified, but their evolution over time will also be tracked. This will be done by searching for commits associated to one such God Component and observing either growth or shrinkage. The analysis of the God components will also help us in understanding the rationale of the developers and how they deal with these components. All these tasks will be performed by having an in depth analysis on not only the components but also on the issues.

2 Exploration

First, we explore our chosen software project, Apache Tika, by means of **requirements analysis**. [5]. Requirements were compiled using the project's public website [1] and an online book about Apache Tika written by the authors [4].

2.1 Tika's High-Level Architecture

This section focuses on describing and explaining the high-level architecture of Tika. All the architecture's components and their interactions are visualized in the simple diagram, see Figure 1.

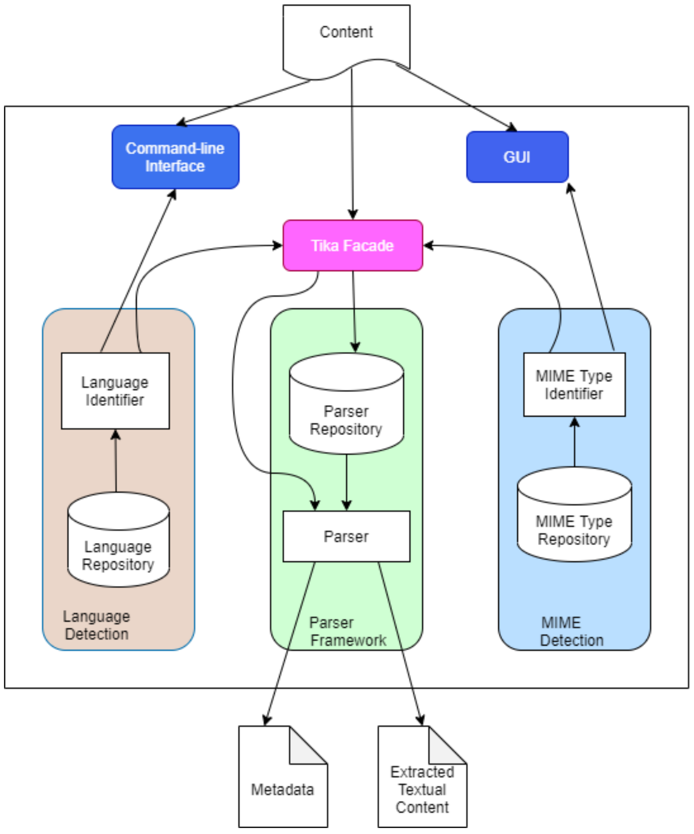


Figure 1: High-Level Architecture. Image composed by ourselves.

The high-level architecture of Tika consists of four key components: the parser framework, the language detection, the MIME detection mechanism and the facade element. All these components play an important role in Tika's overall architecture and they have their own corresponding responsibilities and functionalities. Firstly, the parser framework is the most crucial concept and its main functionality is parsing and extracting all the content and the relevant metadata of any type of document. The language detection component is responsible for carrying out language analysis and this helps in obtaining the metadata of the files. In addition, the MIME detection mechanism is used for detecting and identifying all the available file formats. All these three components contain their own individual repositories, which help in extending Tika by adding or removing new parsers, file formats or language mechanisms. Moreover, Tika's architecture contains a facade, which connects all the main components and it provides a user-friendly frontend for the users that want to benefit from Tika's services. The architecture also consists of some external interfaces like the CLI and the GUI, which allow integrability between users and their applications.

2.2 Key design goals

Let us denote the most important goals the authors set out to accomplish in creating Apache Tika.

2.2.1 Unified parsing

One of the initial key goals set by the creators of Apache Tika was the implementation of a unified parsing interface [4]. In Apache Tika, unified parsing offers a collection of functionalities and a Java interface that deal with external third-party parsing libraries. This was achieved by the creation of the org.apache.tika.parser.Parser interface, which parses the received content incrementally.

2.2.2 Low Memory Footprint and Fast Processing

The second goal of the authors regarding Apache Tika was to achieve a low memory footprint and guarantee a fast processing performance [4]. The creators wanted to make sure that Tika could be easily integrated into any Java application at a low memory cost. This would be beneficial to the users, as they could run Tika in any environment ranging from mobile PDA to desktop computers. At the same time, Tika is expected to react quickly to user's requests and perform file format identification and processing in a fast manner.

As mentioned previously, the received content is parsed incrementally and it is generated as SAX-based XHTML events. SAX (Simple API for XML processing) was the main XML parsing option, as it accomplished all the requirements regarding low memory footprint, fast processing and incremental parsing. In addition, SAX model is more flexible, as it allows its users to decide themselves on how they want to deal with the received content by modifying Tika's parser and specifying what needs to be done atorg.xml.sax.ContentHandlers.

2.2.3 Flexible Metadata

Another design consideration was to ensure the required flexibility that Tika needs in order to perform its tasks on the extracted content [4]. There is an enormous amount of available file formats and Tika is capable of understanding and processing all of them. However, most of these formats contain associated metadata models that provide detailed descriptions about these files. In this case, Tika should also be able to understand the corresponding metadata models. In the previous releases, Tika was always modifying and generalizing the metadata of the extracted content, so that it could fit to its predefined structure. In the latest releases, Tika is not following the same technique and it has become more flexible, as it stores the metadata in their original form.

2.2.4 Parser integration

Another key design goal that was taken into account was the parser integration [4]. Similar to the metadata models and file formats, there are also a lot of parsing libraries and Tika has to easily integrate them within the application. According to the creators, from a design perspective Tika virtualizes the underlying parser libraries and ensures their conformance to Tika's parser interface. However, this is a complicated task as the authors had to deal with parser exceptions, threads of control, delegation, and nuances in each of these libraries. Even though it was a cost-effective process, it brought a lot of benefits like cross-document comparison, uniformity, standardized metadata and extracted text.

2.2.5 MIME Database

Another design consideration was focused on the usage of MIME database, which contains a simple and an efficient way of categorizing the file formats [4]. The main goal of the authors was for Tika to support a flexible mechanism that could in a user-friendly way define and identify different media types. Moreover, Tika contains an XML-based mechanism that is responsible for adding new media types, regular expressions and glob patterns.

2.2.6 MIME Detection

Based on the previous section, the authors continued with making different flexible MIME detection tools available to the end users, like via byte[] arrays, java.io.Files, filenames and java.net.URLs pointing to the files etc [4]. They also focused on making the MIME information available to Tika's parser and metadata, because the detected media type could be an essential source of information to return as extracted metadata along with the parsing operation.

2.2.7 Language detection

Language detection has also become an important feature in Tika [4]. The ability of understanding a language is essential, as it provides useful information regarding the content of the file and its corresponding metadata. Nowadays, the developers are trying to improve language detection in Tika by adding tools that improve language-specific charset detection in the parser.

2.3 Quality attributes

This section contains the identified architectural significant requirements and it is divided into two parts. The first part contains the primary requirements, while the second part includes the secondary quality attributes.

2.3.1 Primary Quality Attributes

1. Flexibility. Flexibility is a significant requirement and it is quite emphasized on Tika's documentation. In general, flexibility represents the capability of a given system to adapt to different environments, settings or to adjust when changes occur. Tika consists of an enormous amount of parsers and detection mechanisms, which can be applied for all the potential user scenarios. This characteristic defines Tika as a quite flexible tool, which can be adjusted to any setting.

   > "First and foremost, we wanted Tika to support a flexible mechanism to define media types..."
   >
   > "By adopting the SAX model, Tika allows developers and those wishing to customize how Tika's Parser deals with extracted information to define custom parsers..."
   >
   > "Provide Flexible MIME Detection:To expose the MIME information programmatically, we decided to expose as many MIME detection mechanisms..."
   >
   > "Beyond that detail (Tika opted to allow one to many types per Parser, achieving the greatest flexibility and decreasing the overall number of parsers), the exchange of MIME information between Parser and Metadata object was another important consideration..."

2. Extensibility. Another significant quality attribute in Tika is extensibility. Extensibility refers to the capability of the system to add new elements or functionalities without negatively affecting the overall performance. In Tika's architecture, the developers have made sure that Tika can be extended by adding new parsers, language detection mechanisms or file format detection tools.

> ”Throughout its architecture, Tika leverages the notion of repositories: areas of extensibility in the architecture. New parsers can be easily added and removed from the framework, as can new MIME types and language detection mechanisms...”

3. Performance. Performance is also another significant requirement in Tika. It demonstrates how the systems reacts while performing given tasks at a certain time. The creators of Tika developed Tika in a way that it could respond fast to requests. Although at the time of its making not similar products might have existed, Tika would have been quickly superseded if it was not performing well enough.

   > “The necessity of detecting file formats and understanding them is pervasive within software, and thus we expect Tika to be called all the time, so it should respond quickly when called upon..”

   > “SAX, on the other hand, parses tags incrementally, causing a low memory footprint, allowing for rapid processing times...”

4. Integrability. In general terms, integrability shows the capability of a component/system to integrate with other components/systems in a way that all these operate properly together. The developers of Tika emphasize the integrability of the tool into the user’s applications and the parser integration within their system.

   > “External interfaces, including the command line and a graphical user interface allow users to integrate Tika into their scripts and applications and to interact with Tika visually.”

   > “Parser integration: Just as there are many metadata models per file format, there are also many parsing libraries. Tika should make it easy to use these within an application.”

2.3.2 Secondary Quality Attributes

- Usability. The program should be usable both in terms of User Experience (UX) and programming interface (API); this means it should have a nice and well-functioning GUI and CLI to use the program and an API to use Tika in your own program code.

  > “By adopting the SAX model, Tika allows developers and those wishing to customize how Tika’s Parser deals with extracted information to define...”

  > “The Tika facade (center of the diagram) is a simple, easy-to-use frontend to all of Tika’s capabilities...”

  > “Tika should be embeddable within Java applications at low memory cost so that it’s as easy to use Tika in a desktop-class environment with capacious network...”

3 System Architecture Analysis

An analysis will be performed using common Software Engineering tooling programs, like *Structure101*. Analysis was done in several steps: (1) compiling the software, (2) create a high-level overview and finally (3) analyze separate packages.

3.1 Compiling Tika

To compile the software, we ran

```
mvn clean install -DskipTests
```

in order to install from source. The program took a little over 13 minutes to compile. An excerpt of the terminal output can be see in Listing 1.

Listing 1: Terminal output after successfully compiling Apache Tika.

```
[INFO] Apache Tika eval ............................. SUCCESS [ 20.264 s]
[INFO] Apache Tika examples ......................... SUCCESS [  5.644 s]
[INFO] Apache Tika Java-7 Components ................ SUCCESS [  1.609 s]
[INFO] tika-parsers-advanced ........................ SUCCESS [  0.079 s]
[INFO] tika-parser-nlp-module ....................... SUCCESS [ 58.152 s]
[INFO] Apache Tika Natural Language Processing ...... SUCCESS [03:35 min]
[INFO] tika-parser-advancedmedia-module ............. SUCCESS [  2.265 s]
[INFO] Apache Tika Deep Learning (powered by DL4J) ... SUCCESS [06:25 min]
[INFO] Apache Tika .................................. SUCCESS [  0.053 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 13:43 min
[INFO] Finished at: 2020-11-29T12:09:58+01:00
[INFO] ------------------------------------------------------------------------
```

3.2 High-level overview

We first ran the program byte-code through *Structure101* [3]. See the composition graph, Figure 2.

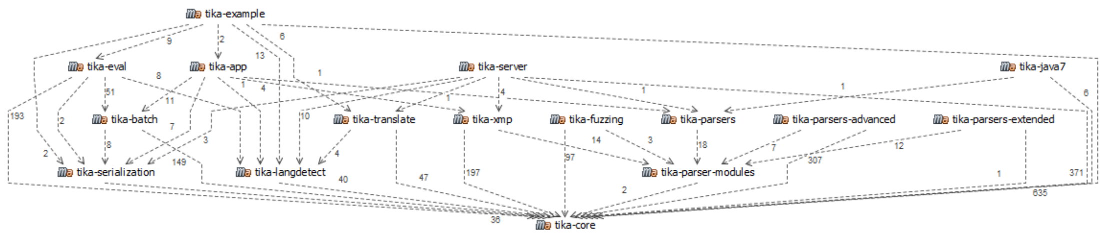


Figure 2: Composition Graph.

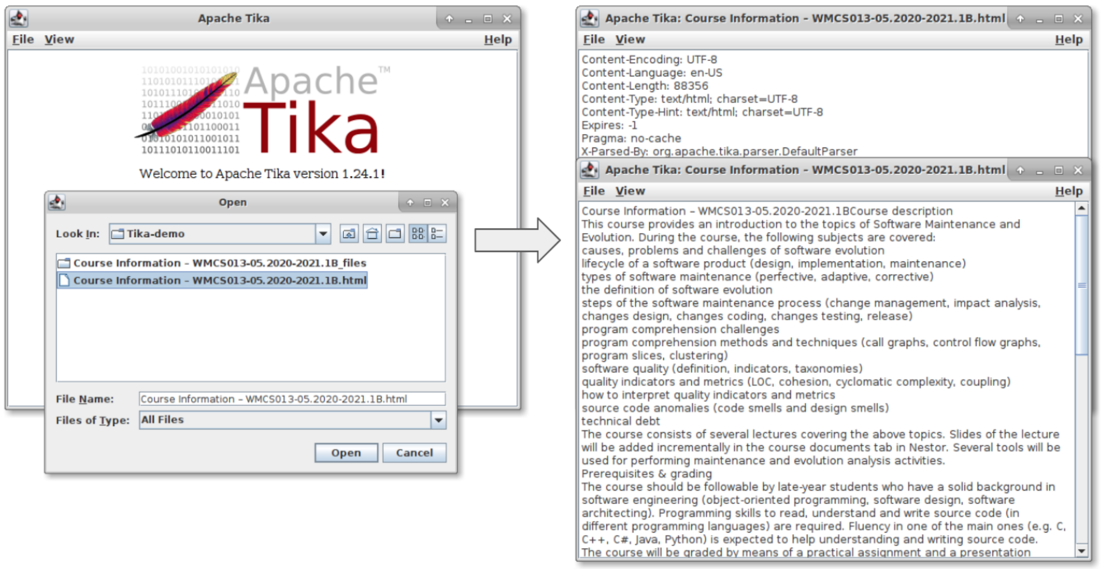
The image (Figure 2) is rather wide, but of high quality - so make sure to zoom in to get more detail if needed. It can be observed that there are 16 main packages in Tika, representing the main functionalities of the code-base. At the bottom of the composition graph we can observe `tika-core`; almost every package depends on it, either directly or indirectly, making it an important package in the code-base. There is lots of inter-dependence between the other packages too, lots of packages also relying on the `tika-serialization`, `tika-langdetect` and `tika-parser-modules` packages. Let's analyze some of the most important packages in more detail.

3.3 Main components

3.3.1 tika-app

The Tika interfaces: CLI and GUI. Allows users to interact with Tika in an interactive way. The GUI encapsulates the core and external parser libraries to build a single runnable jar file, which can be ran on multiple platforms as an user interface or via a command line interface.

To give an example of the tika-app layout and overall capabilities, we ran Tika on a web page of our choosing. The webpage we chose was the Course Information Nestor page for SME, for which Tika extracted meta-data but also tried to determine its main text content. See Figure 3 to see the process visually.



Course Information - WMCS013-05.2020-2021.1B_files
Course Information - WMCS013-05.2020-2021.1B.html

File Name: Course Information - WMCS013-05.2020-2021.1B.html

```
Content-Encoding: UTF-8
Content-Language: en-US
Content-Length: 88356
Content-Type: text/html; charset=UTF-8
Content-Type-Hint: text/html; charset=UTF-8
Expires: -1
Pragma: no-cache
X-Parsed-By: org.apache.tika.parser.DefaultParser
```

```
Course Information – WMCS013-05.2020-2021.1BCourse description
This course provides an introduction to the topics of Software Maintenance and
Evolution. During the course, the following subjects are covered:
causes, problems and challenges of software evolution
lifecycle of a software product (design, implementation, maintenance)
types of software maintenance (perfective, adaptive, corrective)
the definition of software evolution
steps of the software maintenance process (change management, impact analysis,
changes design, changes coding, changes testing, release)
program comprehension challenges
program comprehension methods and techniques (call graphs, control flow graphs,
program slices, clustering)
software quality (definition, indicators, taxonomies)
quality indicators and metrics (LOC, cohesion, cyclomatic complexity, coupling)
how to interpret quality indicators and metrics
source code anomalies (code smells and design smells)
technical debt
The course consists of several lectures covering the above topics. Slides of the lecture
will be added incrementally in the course documents tab in Nestor. Several tools will be
used for performing maintenance and evolution analysis activities.
Prerequisites & grading
The course should be followable by late-year students who have a solid background in
software engineering (object-oriented programming, software design, software
architecting). Programming skills to read, understand and write source code (in
different programming languages) are required. Fluency in one of the main ones (e.g. C,
C++, C#, Java, Python) is expected to help understanding and writing source code.
The course will be graded by means of a practical assignment and a presentation
```

Figure 3: Apache Tika main GUI interface and example of extracting meta-data from a webpage.

Tika managed to get a pretty good idea of the main page content, having extracted the text from the Nestor page correctly. Given just a HTML file, it analyses the page and tries to determine what the most important text is. Note that there are lots of buttons, navigation menu's, and such, that had to be ignored in this case.

Next, to get an idea of the dependency structure of `tika-app`, see Figure 4. See below a description of the dependencies/dependents.

- **Dependencies**: `tika-parsers`, `tika-batch`, `tika-langdetect`, `tika-serialization`, `tika-xmp`.
- **Dependents**: `tika-example`
- **Internal structure**: See Figure 18. Three modules were tagged with a blue dot: `cli`, `gui` and `batch`. These are the main entry points to the module, and are all packed inside `tika-app.org.apache.tika`.
- **Purpose**: This component helps the users to interact with Tika and make use of its functionalities. It contains a simple and an easy to use interface which provides all Tika's capabilities.

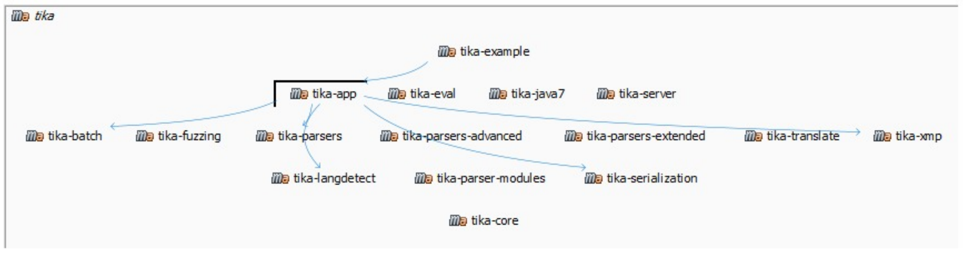


Figure 4: Dependencies of `tika-app` in the overview of all Tika packages.

- **Content**: Three modules were tagged with a blue dot: `cli`, `gui` and `batch`. These are the main entry points to the module, and are all packed inside `tika-app.org.apache.tika`. These external interfaces, like the CLI and GUI, allow the end-users to communicate and use Tika into their applications.

3.3.2 tika-core

This module packs Tika's core functionality. It includes a module to detect file content types, a framework for parsing (text) content from a file and more, like language detection and so forth.

To get an idea of the dependency structure of `tika-core`, see Figure 5. See below a description of the dependencies/dependents.

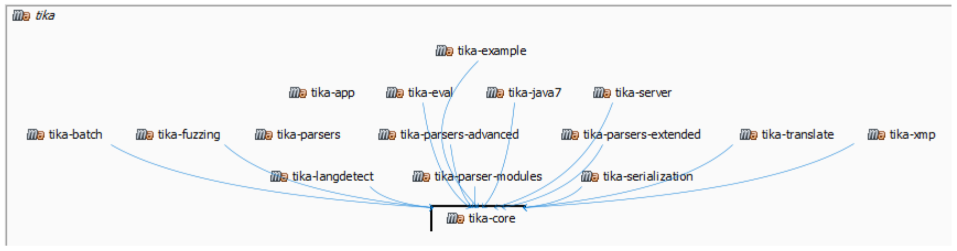


Figure 5: Dependencies of `tika-core` in the overview of all Tika packages.

- **Dependencies**: None
- **Dependents**: Every other package **directly** relies on `tika-core` except `tika-app` and `tika-parsers`. See Figure 5.
- **Purpose**: This component represents the foundation on which other important components are constructed, like Tika app and Tika parsers. It consists of core interfaces and classes.
- **Content**: The core package consists of classes that build Tika facade, the mime package which is used for file format detection and the parser package used for parsing. All the other parser packages present in the overview in Structure 101 extend this package with extra capabilities. In addition, the language identifier mechanism, the metadata package and sax package are present in the core library and they are responsible for language analysis, metadata extraction and outputting structured text, respectively.

3.3.3 tika-parsers

- **Figure**: See Figure 6
- **Dependencies**: `tika-parser modules`
- **Dependents**: `tika-app, tika-java7, tika-server.`
- **Purpose**: According to the documentation, Tika parsers are considered as key features that ensure the main functionalities of Tika. These parsers are responsible for parsing, understanding and processing all the existing file formats and their corresponding metadata models. Even though these tasks are intense and complicated, the parsers camouflage their complexity by providing a really simple tool to the users for extracting the needed text content.
- **Contents**: This component consists of the tika parsers. Tika parsers are fundamental elements of Apache Tika. They contain a set of classes that construct the Tika Parser interface based on external parser libraries.

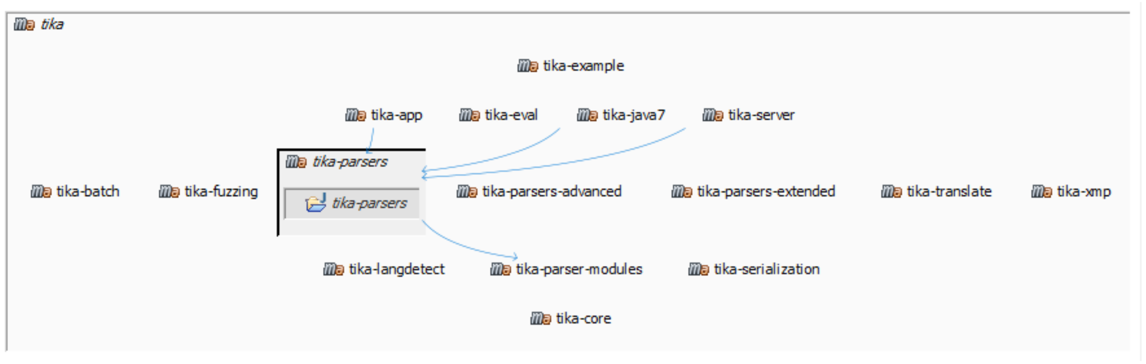


Figure 6: tika-parser component

3.3.4 tika-parsers-advanced

- **Figure**: See Figure 7
- **Dependencies**: `tika-core, tika-parser-modules.`
- **Dependents**: None
- **Purpose**: Tika-parser-advanced is another crucial component that consists of several advanced parsers. These parsers provide more sophisticated functionalities and they are mainly focused on extracting more detailed information from complicated text documents like electronical clinical records, journals etc.
- **Contents**: This component offers extra parser functionalities and it consists of 4 modules. The first module is `tika-age-recognizer` and it is capable of detecting and extracting the ages of people in a given text. The second one is `tika-dl` and its main functionality is image recognition. `Tika-parser-advancedmedia-module` is another parser and it is responsible for captioning and recognizing objects present in images and graphics. Finally, `tika-parser-nlp module` offers functionalities such as: parsing biomedical information, detecting locations in a given text and providing them geo tags based on latitude/longitude, extracting journal information etc.

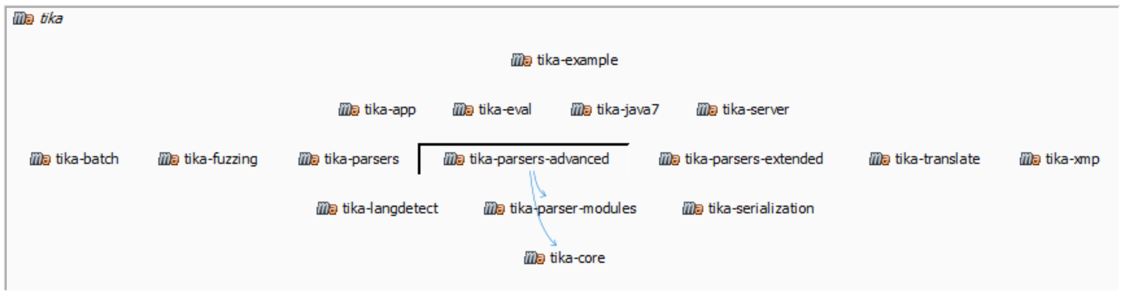


Figure 7: tika-parser-advanced component

3.3.5 tika-parsers-extended

- **Figure**: See Figure 8
- **Dependencies**: `tika-parser-modules, tika-core`
- **Dependents**: None
- **Purpose**: This component is an extended version of the previously mentioned tika-parser components. It consists of additional parsers that are capable extra file formats.
- **Contents**: This component consists of several parsers that provide additional capabilities. The first module is tika-parser-scientific-module and it parses extra file formats like NetCDF files, HDF files, geo file formats etc. The next module is tika-parser-sqlite3-module and it parses SQLite3 files. The last module is tika-parser-extended-integration-tests and it contains integration tests for Apache Tika.

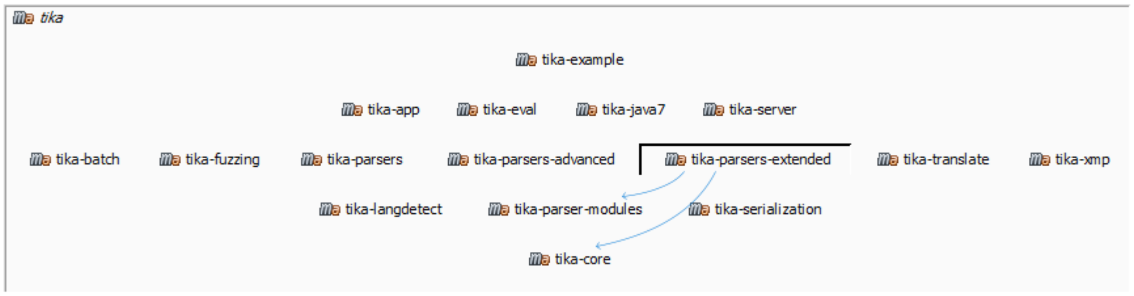


Figure 8: tika-parser-extended component

3.3.6 tika-parsers-modules

- **Figure**: See Figure 9
- **Dependencies**: `tika-core`
- **Dependents**: `tika-fuzzing, tika-parsers, tika-parser-advanced, tika-parser-extended, tika-xmp`.

- **Purpose**: This component provides the basic parsers that are mainly applied in common text file formats like Microsoft documents, PDF files etc.
- **Contents**: : This component contains multiple parser modules. One of these parser modules is `tika-parser-audiovideo-module` which parses audio and video metadata files. Another module is `tika-parser-cad-module` and it is responsible for searching for bits in the headers of the texts. In addition, `tika-parser-crypto-module` is another module that consists of parsers for PKCS7 data and for Time Stamped Data Envelope. There are also other parsers in this module that extract information from Microsoft documents, pdf files, mails, html pages, xml etc.

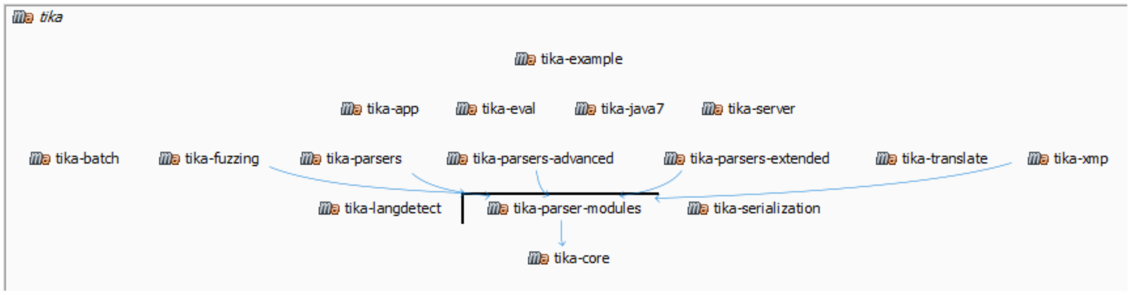


Figure 9: tika-parser-modules component

3.3.7 tika-eval

- **Figure**: See Figure 10
- **Dependencies**: `tika-batch, tika-langdetect, tika-core, tika-serialization.`
- **Dependents**: `tika-example.`
- **Purpose**: The main purpose of this component is to provide insight based on the output of a given extraction tool or to perform comparisons between different tools. In addition, this component helps the developers to make comparisons of the output of two versions of the same tool or their execution while they run on different settings.
- **Contents**: Tika.eval consists of some elements. One of them is `eval.db` which stores the extracted content and the corresponding metadata. Another part in `tika.eval` is `tika.eval.textstats` and it contains interfaces that measure language probabilities and token stats. In addition, `tika.eval.langid` contains the IDs of the languages. Tika.eval provides its own predefined list of reports where users can instantly report insights.

3.3.8 tika-server

- **Figure**: See Figure 11
- **Dependencies**: tika-parsers, tika-langdetect, tika-core, tika-serialization, tika-translate, tika-xmp.
- **Dependents**: None
- **Purpose**: Tika JAX-RS REST application. This is a Jetty web server running Tika REST services.

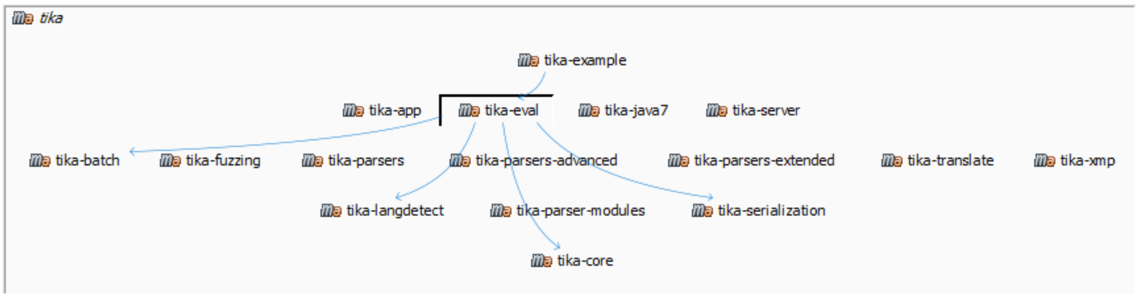


Figure 10: tika-eval component

- **Contents**: This component contains many classes that are responsible for maintaining and monitor the server of Tika. Some of these classes are: **`serverstatus, servertimeout, taskstatus, tikaserverwatchdog`** etc. All the classes of this component make sure that the server is running properly and executes its tasks correctly. In addition, they detect when something is wrong and notify the developers.

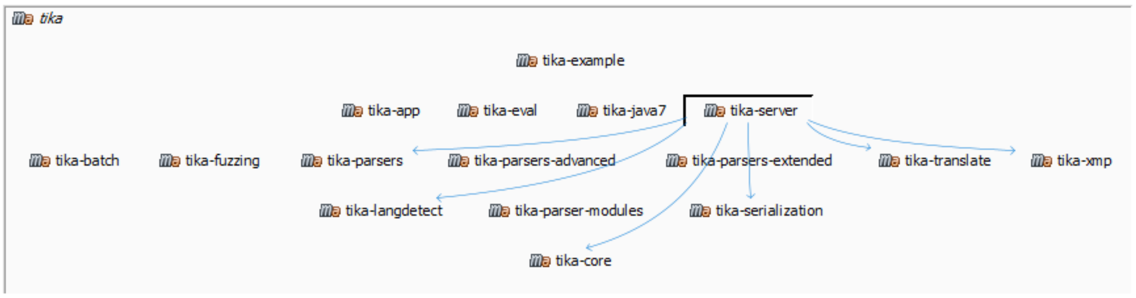


Figure 11: tika-server

3.4 Other components

- `tika-bundle` OSGi bundle that contains tika-core and tika-parsers as dependencies - but no code itself.
- `tika-example`. Contains some usage examples; including those shown on the Usage Examples page. Is neither runnable nor meant to be directly used by another package. Because it contains examples resembling a diverse set of use cases, it is dependent on many other modules, i.e. `tika-app`, `tika-serialization`, `tika-translate`, `tika-langdetect-optimaize`, `tika-eval-core` and `tika-core`.
- `tika-fuzzing`. Contains very little documentation. Contains a `AutoDetectTransformer` class that seems to take in an array of transformers and allows one to work with the entire vector of transformers at once.

- `tika-xmp`. Provides a "*conversion of the Metadata map from Tika to the XMP data model*". XMP is a format for storing file metadata in a consistent way. The module contains a bunch of classes that facilitate XMP conversion from different formats, like OpenDocument, MSOffice and RTF. Is dependent on `tika-core` and some `tika-parser-` modules for processing Microsoft Office documents.
- `tika-serialization`. Contains some serializers and de-serializers for storing and retrieving metadata as JSON objects. Is dependent on `tika-core`, but really only uses interfaces from the package instead of actual functionality, namely `org.apache.tika.metadata.Metadata`; used working with Metadata objects in the module.
- `tika-translate`: It represents an interface that provides translating services and it consists of multiple translators. One of these translators is MosesTranslator that uses Moses decoder for translations. Another translation service is CachedTranslator and it is responsible for saving a map of previous translations in order to not repeat translation requests. In addition, this component consists of YandexTranslator that provides a REST client implementation of Yandex Translate API. Among these translation services, this component also makes use of GoogleTranslator and MicrosoftTranslator.
- `tika-langdetect`: This component is capable of identifying the language of a given text. This type of information is quite helpful as a lot of metadata might not provide the language of the text formats. This component consists of a few language detectors like langdetect.lingo 24, langdetect.commons etc.
- `tika-batch`: This module has the purpose of conventional processing and it has a producer/consumer design pattern. This module is still under development and the developers try to keep it as configurable as possible. It consists of a ResourceCrawler, which adds potential files for processing onto the queue. A ResourceConsumer is also present in the batch and it pulls a resource from the queue and consumes it. In addition, a StatusReporter often reports on how many files have been processed etc.

4 Identifying God Components

To identify God Components, let us remind ourselves of the definition of a God Component. It is a software component whose Lines Of Code or number of classes got so extraneously large that the code got unmanageable. To find such components in our software, we need a tool to parse the entire code-base and infer the appropriate statistics from them. Probably, a tool written in Java would suit best, since it could use native methods to parse the code-base and thus reverse-engineer the software in an efficient way. A tool that is well suited for our purpose is *Designite* [9]. Designite is a quality assessment software tool that can be used to identify numerous technical debts in your software codebase. Among which, architectural smells and including God Components. Let us explain how we used this software to find God Components in Apache Tika, as well as fetch any additional data we need for a comprehensive analysis.

4.1 Designite

Designite defines God Components in terms of lines of code and number of classes. The higher the amount of classes or lines of code, the more chance a component is considered as a God component. Designite classifies packages as a God Component if the package has more than 30 classes, or due to some other predefined condition which is less commonly encountered.

To test Designite functionality, we ran the tool on the latest version of the Tika git repository; at the time of writing `2.0.0-SNAPSHOT`. Having ran the Designite Enterprise edition, 15 God Components were identified. See Listing 2 for an example of Designite's analysis summary.

Listing 2: Terminal output after running Designite on Tika at the `2.0.0-SNAPSHOT` version, commit `5a27811`.

```
git:(master) $ java -jar src/runtime/DesigniteJava_Enterprise.jar
        -i ~/git/tika -o tika
Searching classpath folders ...
Parsing the source code ...
Resolving symbols...
Computing metrics...
Detecting code smells...
Exporting analysis results...
--Analysis summary--
        Total LOC analyzed: 125928      Number of packages: 155
        Number of classes: 1568 Number of methods: 10541
-Total architecture smell instances detected-
        Cyclic dependency: 60   God component: 15
        Ambiguous interface: 0  Feature concentration: 72
        Unstable dependency: 16 Scattered functionality: 0
        Dense structure: 1
-Total design smell instances detected-
        Imperative abstraction: 8       Multifaceted abstraction: 7
        Unnecessary abstraction: 28     Unutilized abstraction: 797
        Feature envy: 0 Deficient encapsulation: 169
        Unexploited encapsulation: 0    Broken modularization: 25
        Cyclically-dependent modularization: 8  Hub-like modularization: 1
        Insufficient modularization: 72 Broken hierarchy: 5
        Cyclic hierarchy: 0     Deep hierarchy: 0
        Missing hierarchy: 0    Multipath hierarchy: 0
        Rebellious hierarchy: 0 Wide hierarchy: 0
-Total implementation smell instances detected-
        Abstract constructor func call: 1     Complex conditional: 227
        Complex method: 324     Empty catch clause: 361
        Long identifier: 107    Long method: 30
        Long parameter list: 172        Long statement: 1236
        Magic number: 3739      Missing default: 38
----
Done.
```

We can see that the codebase contains 125,928 Lines of Code (LOC) and has 10,541 methods spread over 1,568 classes packed up in 155 packages - among which there exist 15 God Components. Designite outputs its detailed analysis results to `.csv` files in a specified folder, in this case `/tika`. Among the .csv files is `ArchitectureSmells.csv`, which is the file we need. We can easily read the file and extract only the information important to us: the rows concerning God Components.

Note that the above report only applies to the `2.0.0-SNAPSHOT` version of the code at commit `5a27811`: the challenge now lies in running Designite for **all** versions of the code. More precisely said, the aim is to run Designite for every **commit** of the code - enabling us to analyze the evolution of God Components in the finest resolution possible. Let's build a script that runs Designite on every commit for us, automatically.

4.2 Running Designite programmatically

The relevant files for our Designite data-collection code are `find_gcs.py` and a support file, `git_utils.py`. Like their name imply, they are both Python scripts, where `find_gcs.py` is dependent on the `git_utils.py` module. Let us describe the step-by-step process of the entire operation.

1. **Grabbing commits**. First, we need a list of commit ids (SHA-1 hashes) such that we can devise check out the specific version of the code one by one and run Designite on it. We do this, by running a `git log` command, with some `--pretty` modifiers to output the information in a desirable format. We then use the Python Pandas [7] module to read in the information into a *DataFrame*, a tabular datatype for storing large datasets. Columns that get stored include commit id, author name, commit date/time, commit message and the relevant Jira issue, if present in the commit message. The described functionality is described in the GET_COMMITS function in `git_utils.py`. Examples of some rows are as seen in Table 1:

id	author	datetime	message	jira
7f65d61	THausherr	2020-12-14	TIKA-3248: revert accidental commit ..	TIKA-3248
326b7d7	THausherr	2020-12-14	Revert "Merge origin/main into main"	NaN
dd85c73	THausherr	2020-12-14	Merge origin/main into main	NaN

Table 1: Commit data as mined by our application, stored in a Pandas Dataframe. Commit **time** was omitted from the 'datetime' column for brevity.

   Note there also is an example of the resulting DataFrame in the `statistics.ipynb` Notebook in the root folder of our repo.

2. **Running Designite**. Next, we loop each commit and run Designite from the Python script using `os.system` - a function used to execute terminal commands. Once Designite finished running, it outputs `.csv` files to some chosen directory. We take the relevant file (`ArchitectureSmells.csv`), extract the God Components, and again output that information to a .csv file of our own. The data also has its columns renamed and some transformed for easier data processing down the line: the reason why Designite classified the component as a GC is extracted, for example, along with the relevant metric, like # classes. See Table 2:

commit	repo	package	smell	cause	metric
49bb469	tika-cpu_21	org.apache.tika.example	God Component	MANY_CLASSES	49
49bb469	tika-cpu_21	org.apache.tika.batch	God Component	MANY_CLASSES	31
49bb469	tika-cpu_21	org.apache.tika.detect	God Component	MANY_CLASSES	31

Table 2: Designite output data, captured per-commit and with the '# classes' metric processed into a separate column.

   The main reason Designite classified components as GC's, is the MANY_CLASSES cause (see Table 2). In fact, throughout all commits, this was the only reason that Designite classified components as a GC. This fact can simplify the analysis process later - we could assume every row to have a `metric` property indicating the 'large' amount of classes that made it a God Component. So, let's now assume the reason to always be MANY_CLASSES, allowing us to utilize the metric that is output alongside the report: we rename the `metric` to `# classes`. Note, that Designite classifies a component as a GC when # classes > 30.

   Also, for memory purposes, we remove the other, irrelevant Designite reports that are concerned with other bad architectural smells. The relevant code for this step is found in `find_gcs.py`.

We are now able to run Designite for all commits in the Tika project! That's great. But, because the Tika repository has nearly 5,000 commits, we would have to run Designite around 5,000 times. With an average runtime of about 40 seconds, it is easy to see that the computational complexity got quite large and the problem would take quite a while to run on one CPU, around 55 hours of running Designite non-stop. This might be a bit infeasible to do on just simple Laptops. Therefore, we utilized the University's High-Performance Computing cluster *Peregrine* to solve the problem.

4.3 Running Designite on Peregrine

Peregrine is a HPC cluster containing computational nodes for several purposes. Our goal is to speed up the Designite running process, which requires a working copy of the Tika repository to check out a certain version (commit) of the code. On Peregrine, we have access to at least 24 CPU's in a single node. The idea we implemented was to run Designite simultaneously on each of the 24 CPU's available to us in the HPC environment. This also means, however, that we also need 24 working copies of the Tika repository. Dependent on the exact Tika version checked out, the repository comes in at around 2 GB. During the process of running Designite, the entire code-base will be loaded up into memory. So, theoretically this would add up to around 48 GB. In practice, however, much more memory is needed to load up 24 copies of Tika in Designite at the same time: there's probably some memory overhead in storing certain Data types and making the necessary computations and inferences. We therefore fall short on memory when utilising a 'regular' Peregrine node, which has 128 GB memory available. Luckily, however, Peregrine has '**high memory**' nodes available to us.

The high memory nodes allow us to set up to 1,024 GB (1 TB) of working memory, available to all our cores. Using this gigantic amount of memory, we were able to run Designite on 24 versions of the code simultaneously. Even though the queue time for this node is quite long, we only have to go through this process once, if we do it well. So let's see how we architected the parallel structure in the code.

On every run, the commits that had no Designite run yet, are mapped over the available CPU's. Then, the CLONE_TIKA function in `git_utils` makes sure that every CPU has its own clone of the Tika repository. To make sure each run of Designite is ran on a clean version of the repo, `git clean` is used, among others. Having ran the entire operation on Peregrine, we receive a log output file that has the summary as seen in Listing 3.

Listing 3: Peregrine job output.

```
Peregrine Cluster
Job 16174792 for user 's2995697'
Finished at: Wed Dec 16 02:40:31 CET 2020

Job details:
============

Job ID              : 16174792
Name                : sme_god-components
User                : s2995697
Partition           : himem
Nodes               : pg-memory05
Number of Nodes     : 1
Cores               : 24
State               : COMPLETED
Submit              : 2020-12-15T10:17:59
Start               : 2020-12-15T21:08:07
End                 : 2020-12-16T02:40:29
Reserved walltime   : 2-22:00:00
Used walltime       :   05:32:22
Used CPU time         : 5-10:36:50 (efficiency: 98.25%)
% User (Computation): 91.59%
% System (I/O)        :  8.41%
Mem reserved          : 1000G/node
Max Mem used          : 124.29G (pg-memory05)
Max Disk Write        : 153.60K (pg-memory05)
Max Disk Read         : 7.65M (pg-memory05)
```

It can be observed that our job ran with high efficiency (98.25%), indicating we put all nodes at work nicely. The maximally used amount of memory comes in at 124 GB, indicating we only barely topped out the 128 GB memory available to us on regular Peregrine nodes. Apparently, no memory can be allocated already before the job report says to have used 128 GB memory. In all, it took the job more than 5 hours to complete, giving us a nice speedup on the sequential operation.

That said, once this massive operation is complete, we simply take all the separate .csv files containing the analysis results and concatenate them into one huge result file. This file is stored in `output/all_reports.csv`.

4.4 Fetching Jira issue data

Next up, we need to fetch data from the Jira issue tracker, such that we can utilise this data in our analysis later. Contributors to Tika use the issue tracker to track open issues, assign developers to them and discuss about the issue in a comment thread. It functions much like the Github issues tool. Now, our goal is to fetch the additional information on the issues that is valuable to our analysis: the issue *type*, the internal components affected, etcetera. Let us first note, that we have parsed the tika **issue key** from the commit message using a regex, if it was mentioned in the commit at all (see the 'jira' column in Table 1).

Using this key, we are able to fetch issues using the public API. Apache hosts a version of Tika itself, meaning we have to fetch from the `issues.apache.org/jira` domain instead of from `jira.atlassian.com`. We fetch 50 issues at a time and sequentially keep fetching until we reached the total amount of issues. The file used to do this is `jira_issues.py`. Among many other columns, an excerpt of some issue data can be seen in Table 4.4.

jira	status	creator	issuetype	created
TIKA-3256	Resolved	tilman	Task	2020-12-27T13:07:15.000
TIKA-3255	Open	peterkronenberg	Bug	2020-12-22T17:04:55.000
TIKA-3254	Open	sathia	Bug	2020-12-22T13:49:17.000

Once completed the data is stored in a `.csv` file, `/output/all_issues.csv`.

4.5 Computing Lines Of Code changed

Another interesting data source that helps in investigating the evolution of God Components is the **size** of the God Component, which can be represented in both number of classes, or the total Lines Of Code (LOC) of the component. Note that for the former, we have already acquired the data required: Designite returns the amount of classes along with its report, as can be seen in Table 2. For the latter, however, we require some additional computations.

The goal is to compute the Lines Of Code for every commit and every God Component. This means that before we start, we need to load up the data computed in the previous steps which describes the God Components for every commit of the code. Given that information, we can iterate every commit and run a git command, `gif diff --numstat  <commit-id> <commit-id>`, which gives us the additions and deletions to every file affected in the given commit.

The computation is running using the COMPUTE_LOCS function in `git_utils.py`. Like said, COMPUTE_LOCS uses `git diff` to discover what files changed for some certain commit. The output is then formatted and loaded up into a Pandas data frame for further analysis. For every file changed in some commit, we found out whether it affected some God Component; we do this by checking whether the package name exists in the path. An example match would be `tika/tika-core/src/main/java/org/apache/tika` ⇔ `org.apache.tika`. Note we prefix the God Component package name with `src.main.java` to only match the main source code, i.e. not the testing code. Moreover, packages that have higher levels of nesting are tested on a match first, i.e. `org.apache.tika.parser` will be tested before `org.apache.tika` because its nesting level is higher. Only the first match is taken into consideration - preventing multiple packages matching just one file. For an example of how this processed data looks like, see Table 3:

additions	deletions	godcomp	id	author	...
1	3	org.apache.tika.batch	7f65d61	THausherr	...
8	14	org.apache.tika.parser	7f65d61	THausherr	...
2	3	org.apache.tika.parser.microsoft	7f65d61	THausherr	...

Table 3: Lines Of Code added/removed information captured per-commit, for each God Component that was found to have its files modified by the commit. Note some columns relating to the git commit were omitted for brevity sake, like the commit message.

With all per-commit info from git diff mapped to a God Components, we can map the total Lines Of Code (LOC) that each commit builds up to each God Component. Before we aggregate the data, however, we store the data in full in a csv file, `/output/all_locs.csv`.

4.6 Data analysis

Now finally that we have a huge data file, we can to analyze it. We do this in the a Jupyter Notebook called `statistics.ipynb`. The Notebook is built such to not include complicated logic itself, but rather just load data and then aggregate and visualize it in human-readable format. Notebooks lend themselves well for such data analysis purposes, because developers are able to include rich-text comments in the file itself, as well as show image right in the Notebook. The Notebook is available in our Github repository, but we also hosted it as a web page ourselves, and is accessible at: https://dunnkers.com/god-components.

4.6.1 Computing the chronological class difference

One (important) computation that is worth noting, is the *# classes chronological difference* (delta) computation. In this computation, we compute the number of classes each commit 'added' or 'removed' to some God Component. To do this, we line up the known amount of classes for every God Component chronologically, that is, by sorting the commits by date. Then for every commit, we compute the **difference** between that current row and its previous one. We could express this easily in some maths as as a function of $classes(C_i)$, the amount of classes given some commit C_i, i.e.

$$\Delta classes(C_i) = classes(C_i) - classes(C_{i-1}) \qquad \Delta classes(C_0) = 0 \tag{1}$$

where C_i is commit i in a list of commits ordered by date time, such that C_0 is the first commit in Tika and C_n is the most recent one, i.e. $i \in [0, n]$. Given this notation, C_{i-1} means the commit before the current index (time point) i.

We are also interested in the amount of classes that were **added** or **removed**, specifically. This is now easily computed using the following operations:

$$classes_{added}(C_i) = max\{\Delta classes(C_i), 0\} \tag{2}$$

$$classes_{removed}(C_i) = min\{\Delta classes(C_i), 0\} \tag{3}$$

Using these simple equations we can extract interesting information from our dataset; we are able to see exactly which developer, in which commit, added or removed classes to some God Component. Let us now, using all the data obtained in this chapter, do a **quantitative analysis** to find out how God Components evolve in Apache Tika.

5 Quantitative analysis of God Component evolution

Now that we have all the data we need, we can perform a statistical analysis. Let us remember that the goal is to uncover the evolution of God Components, i.e. find out when God Components are created or removed, by whom, how big they are, et cetera. In this section we will attempt answering questions using data only; in a **quantitative** way. We pose ourselves the following questions:

- How long do God components stay inside a system? (Section 5.2)
- How many developers contribute to God components? (Section 5.3)
- What types of issues contributed to the creation of God components? (Section 5.4)
- What types of issues contributed to the re-factoring of God components? (Section 5.4)

All of which provide us with insight regarding the evolution of God Components in Tika. But before we start answering these key questions, let us warm up with some general statistics on the data.

5.1 General God Component statistics

First, we line up what God Components were found using Designite at all - and how long they've been in the code base for. See Table 4.

It can be observed in Table 4 that 15 God Components were found throughout Tika's entire history. All God Components that were found were still a God Component to the date of writing. The found God Components do vary in total age; some only just first appeared very recently (`org.apache.tika.utils`), whilst others have been around for over a decade (`org.apacke.tika.parser.txt`).

Note that for the # authors, it is important to comment that the number might be slightly skewed. In git, a single author might use multiple computers having varying aliases or email addresses registered in his/her git configuration. To counter this duplicate-author issue, git has something called a *mailmap*, which allows mapping duplicate authors back to a single name. We used our own method of mapping, using a simple mechanism using the csv file at `output/authormap.csv`. Either way around, the duplicate names have to be identified manually. For this reason, we were not able to create a perfect mapping; although our results are probably quite accurate using the *authormap*-corrected data.

Before we turn to the questions, let us first explore the 'size' of our found God Components using two different metrics. We do this in two ways: (1) in terms of # classes versus time per God Component and (2) in terms of Lines Of Code versus time per God Component.

5.1.1 God Component growth in terms of # classes

Since we have a dataset which includes the # classes for each God Component at **every** point in time, we are able to create a neat visualization exploiting this data. This can give us an idea as to how We can visualize the God Components in terms of '# classes' versus the time, see Figure 12.

package	Became GC at	# commits	# days	# authors
org.apache.tika.batch	2015-06-28	2352	1996	100
org.apache.tika.detect	2017-01-19	1579	1424	71
org.apache.tika.example	2015-05-04	2433	2050	101
org.apache.tika.fork	2018-05-31	738	927	45
org.apache.tika.metadata	2016-09-26	1743	1539	77
org.apache.tika.mime	2015-05-02	2443	2053	101
org.apache.tika.parser	2015-02-21	2429	2123	102
org.apache.tika.parser.microsoft	2011-11-25	3052	3306	105
org.apache.tika.parser.microsoft.chm	2020-08-21	155	114	11
org.apache.tika.parser.microsoft.onenote	2019-12-16	305	363	22
org.apache.tika.parser.microsoft.ooxml	2017-03-23	1454	1361	69
org.apache.tika.parser.txt	2009-05-22	4539	4223	107
org.apache.tika.sax	2011-09-25	3691	3367	105
org.apache.tika.server	2014-05-07	1078	2412	38
org.apache.tika.utils	2020-10-31	86	44	7

Table 4: Summary information of all found God Components in the Apache Tika codebase, across all different commits on the *main* branch. All God Components are still a God Component to this day. The column '# commits' denotes the amount of commits made to that God Component in its entire lifetime, i.e. even before being a God Component. '# days' denotes the amount of days that component has been a God Component for, in total (i.e. the component may have stopped being a GC and then again continued to be a GC over its lifetime). Data applies to commits up till 14th of December 2020.

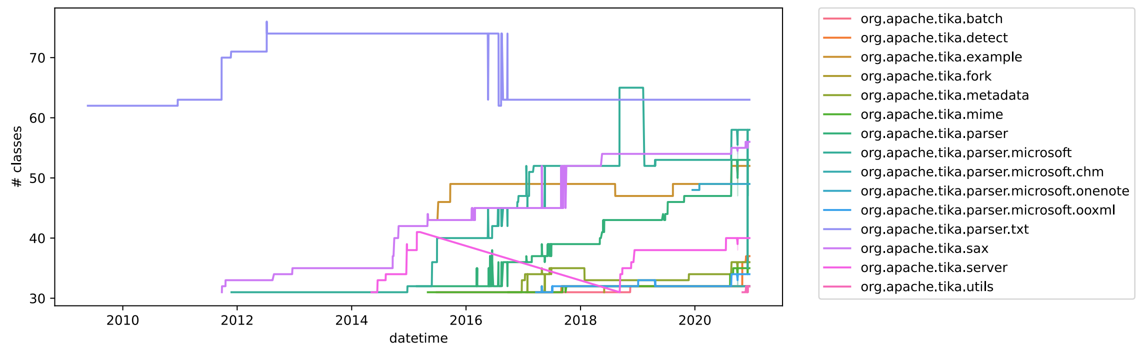


Figure 12: God components growth in Apache Tika. X-axis represents the time and Y-axis represents the number of classes.

Some observations can be made according to the chart. The `org.apache.tika.parser` component first starts out as a GC and remains to do so until now, with a high number of classes, around 60, but fluctuating. A general trend that can be spotted is the components to mostly increase in amount of classes over the time; this seems to apply for almost all GC's. Lastly, we should note that, albeit all GC's have continuous lines over the plot, there might be sections when the line just 'bridges' the gap of the component **not** being a GC (e.g. `tika.server`); the '# classes' data is only available when the component is actually a GC.

5.1.2 God Component growth in terms of Lines Of Code

What would also be interesting, however, is to discover what commits added/removed code to some God Component - and specifically, how the cumulative Lines Of Code for some God Component progresses over time. A computation is done in `git_utils` to compute the Lines Of Code for every commit at every point in time, which data we can exploit in `statistics.ipynb` using a mapping operation to compute the cumulative LOC's over time space. See Figure 13 for the result.

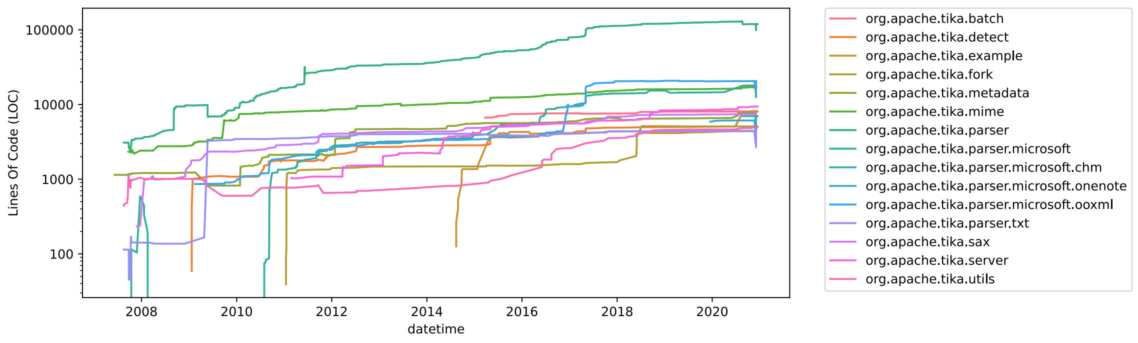


Figure 13: God components growth in Apache Tika. X-axis represents the time and Y-axis represents the lines of code, in **logarithmic scale**.

Note that the y-axis is in logarithmic scale: the differences in LOC buildup between the various GC's was too large to make a nice chart otherwise. It is important to take this fact into account when interpreting the chart - in terms of Lines Of Code, by far the biggest component is `org.apache.tika.parser`. When carefully inspecting the chart, it can be seen that that component is really an order of magnitude larger than the other ones, i.e. others compare at around 10,000 LOC at the time of writing, compared to the parser's around 100,000 lines. That's a big difference.

In the next sections, let us turn to the 4 questions that were posed in the beginning of this chapter.

5.2 How long do God components stay inside a system?

To figure out how long God Components stay inside the system, we iterated every commit and figured out what God Components exist in that version of the code. It is important, to not make assumptions about the starting- and ending dates of the God Components; a component might become a GC and stop being one multiple times. Having aggregated all this data into a suitable data frame, we obtain Figure 14.

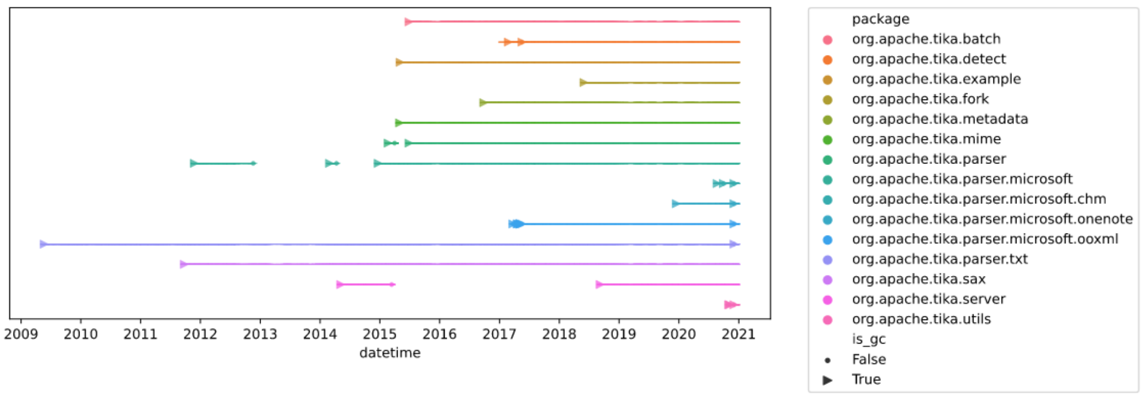


Figure 14: All God Components shown such that all their GC 'starting'- and 'ending' dates are visible; when they became a GC and when they stopped being one.

Interesting insight. Like shown in the overview table earlier, it can be seen that there were in total 15 God Components detected throughout the existence of the Tika codebase. Some have stopped being a GC in between, having a break of sometimes up to a couple years before becoming a GC again (tika.server). However, we should note that some GC's have shorter breaks, sometimes so short that the line in the plot is still continuous - the re-emergence of the GC in that case can be seen by the caret indicator (▸).

There are 8 packages in total that have evolved as God components and continued as such without interruptions. Some of these God components are parser.txt, tika.detect, tika.batch and tika.metadata. Most of these God components were created in nearly 2015 and some others in nearly 2017, implying that they have been in system for approximately 5 years and around 3 years, respectively. However, the system contains some God components that stopped being as such for a while and that eventually ended up again into God components. Some of these God components are tika.parser and tika.server. Tika.parser was defined as a God component in 2012, stopped being one in 2013. This package became a God component again in 2014 and remained as such only for a very short time (a few months). Then eventually, it became a God component again in 2015 and continues to be one up to now. These phases imply that the God component has been present in the system for nearly 6 years. In addition, tika.server was created halfway 2014 and remained as such till 2015. The package was redefined as a God component halfway 2018 and it continues to be one up to now, meaning that it exists in the system for nearly 4 years.

5.3 How many developers contribute to God components?

Given the data we have, the question is relatively easily answered. Let us remember ourselves on our 'delta' data from Section 4.6.1: this data records the # classes that were added or removed given every commit to every God Component in Tika. Using this data, we are able to find out which developers added or removed the most classes to Tika: indicating what developers 'contributed' most to God Components by either adding or removing complexity that directly leads to Designite classifying it as a God Component.

Note that, like said in Section 5.1, the amount of authors data we obtained might have been slightly skewed - but given our *authormap* merging fix it should be workable enough to draw some consistent conclusions. We were able to isolate the developers that had the highest percentage of contribution to the class changes in Tika: both class additions and removals are considered. We took the top 8 contributors that changed the most classes in the code-base: which account for at least 87% of the total class changes. Without furhter ado, see Figure 15.

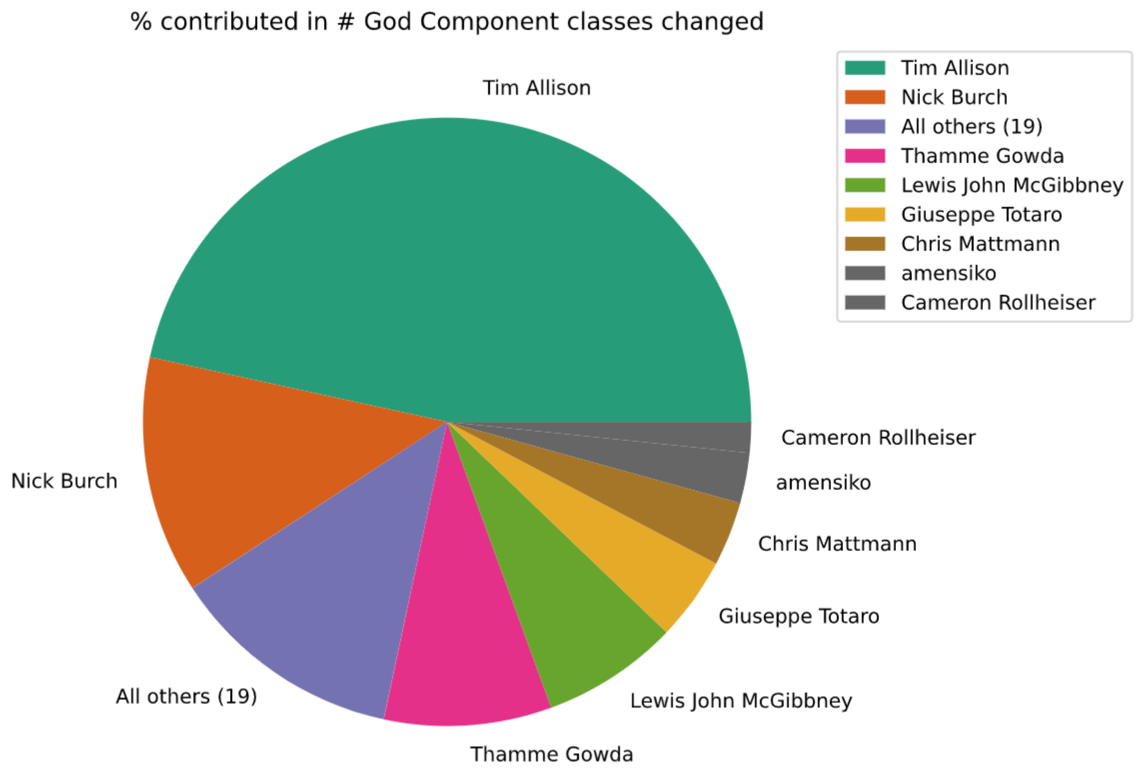


Figure 15: Pie chart showing what developers contributed what percentage of the total classes changed in God Component. Only the top 8 contributors are shown; the others are easily summed because they are a negligible fraction of the total.

As can be seen in the chart above, Tim Allison is the developer with the highest contribution for class changes in Tika, covering almost half of the total contributions (note that Tim Allison commited code under various names - but this data was merged using our *authormap*). Other developers that have had a significant contribution in Tika are Nick Burch, Thamme Gowda and Lewis John McGibbney, with approximately 30 percent of the total contribution. To make our analysis more comprehensive, we can also include a chart that clearly shows both the additions **and** removal of classes to God Components for each of the top 8 developers and all others, see Figure 16.

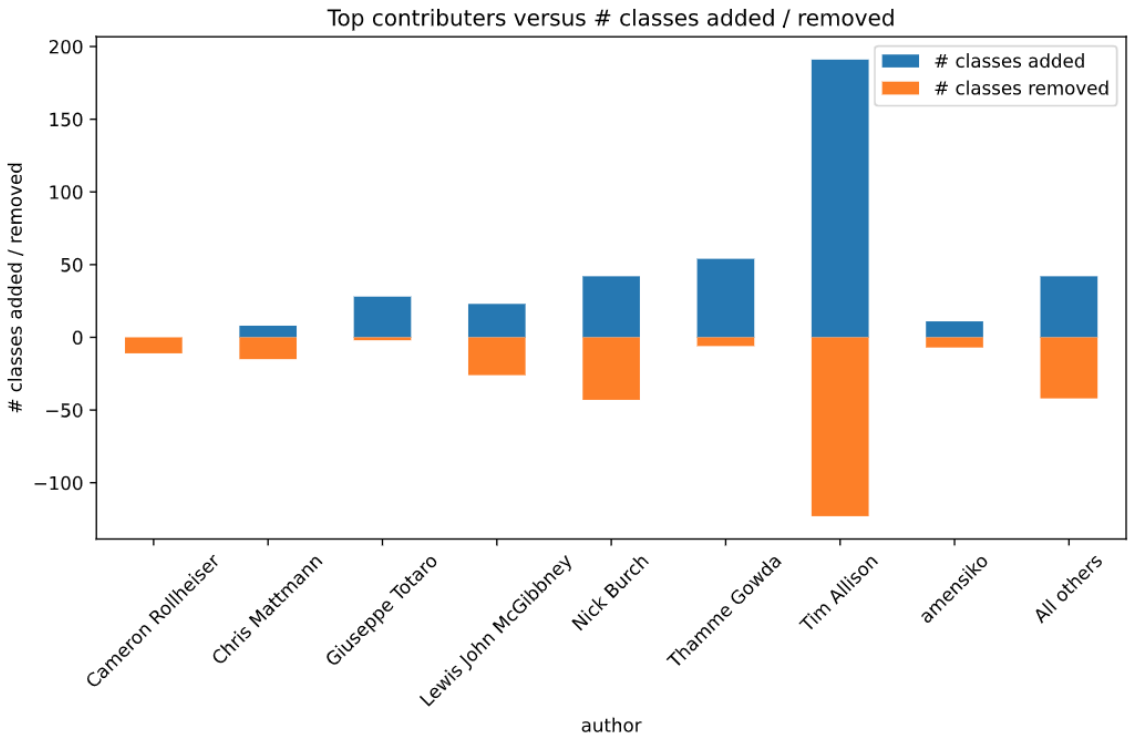


Figure 16: A bar chart showing what 'top' developers added- or removed the most classes.

It can be observed in this chart that among all developers, Tim Allison added a bit more than he removed, but Thamme Gowda and Giuseppe Totaro have an even higher such 'addition' ratio; they remove almost no classes whilst only adding ones. It can be said that they are God Component **creators** but not '**refactorers**'.

Concluding, it can be said that the total buildup of God Components can be attributed to a handful of developers: the top 8 is enough to explain more than 87% of God Component buildup.

5.4 What types of issues contributed to the creation of God components?

The Jira issue tracker classifies the issues of Apache Tika into 7 main categories: bugs, improvements, new features, tasks, sub-tasks, wishlist and testing. All issues in Jira are accompanied with a *key*, which is also apparent in most commit messages. This means we can extract the key from the commit messages - which we did - like explained in Section 4.2. The next step, then, is to join a couple of datasets - the Tika commits, the information on the God Components for every commit, and extra information on every Jira issue. Having combined all this data, we are able to identify what issues contribute most to the creation of God Components. We were able to build a chart that shows both God Component **buildup** in terms of # classes added and **refactoring** in terms of # classes removed, see Figure 17.

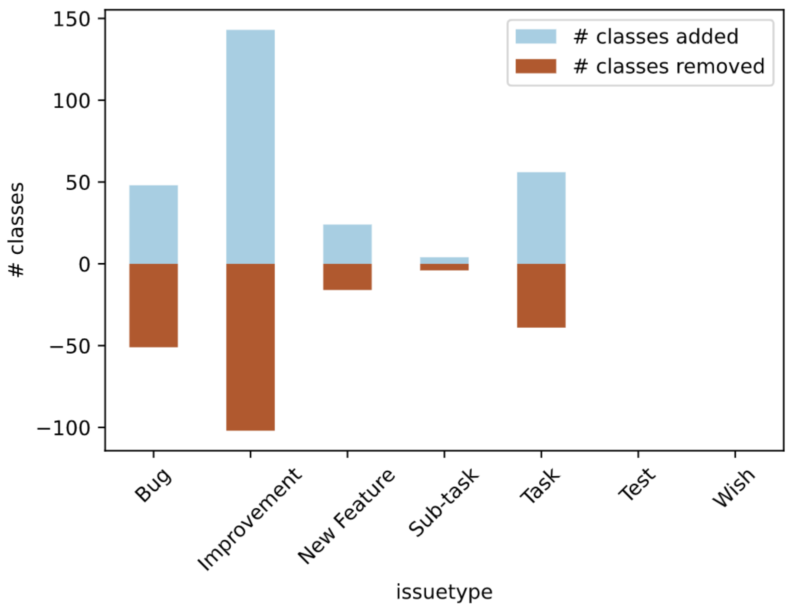


Figure 17: Stacked bar plot of what issues contributed most to the God Component buildup, i.e. issues that led to adding or removing classes to God Components.

Using Figure 17 we can identify that there are three main types of issues that mainly contributed towards the creation of the God components: bugs, improvements and tasks. However, the type of issue that had the most impact in the creation of the God components is the **improvement** type of issue. The image shows that the improvements have added the highest amount of classes, approximately 150 classes, but at the same time they have removed nearly 100 classes. The second type of issue that has had significant impact on the creation of the God components are the bugs, which added around 50 classes, but also removed roughly 50 classes. The type of issues with the least impact on the God components are the wishes and the tests, which have neither added nor removed any classes in the code.

5.5 What types of issues contributed to the re-factoring of God components?

Like before, we analyse data from the the Jira issue tracker. This time, however, we are interested in the **re-factoring** of God Components instead of the creation. Because we built one plot showing both creation and re-factoring of God Components according to issue type, we again refer to the same plot in this question; which is Figure 17.

Based on the analysis conducted upon a set of issues/commits, it has been noticed that the main type of issue that has contributed towards the re-factoring of the God component is the improvement type of issue. In some of these improvements, the developers focus on re-factoring the God components by restructuring modules and packages in a more efficient way, by removing unnecessary dependencies and upgrading components to newer versions. The re-factoring process leads to adding more classes or to removal of classes based on the issue itself. One example of a refactoring issue is TIKA-3179. The issue is an improvement type of issue and is of critical importance. It mainly affected the parsers and the issue was concerned with cleaning the parser's hierarchy and restructuring the entire components. According to Jira comments, a lot of parsers were moved to new packages and the parser.extended package was also created for this issue. These cases have been analyzed more in depth in Section 6.

6 Qualitative analysis of God Component evolution

Now that we have analysed the evolution of God Components in a quantitative manner, let us now take a **qualitative** approach to the problem. For this, we manually analyse issues from the *Jira issue tracker*, which Tika uses as a project management tool. This section focuses on analyzing the Tika's issue tracker. The issue tracker contains approximately 3000 issues. The main goal of the analysis is an attempt to better understand the rationale of the developers regarding their consideration towards God components. In order to achieve this, we have selected a sample of issues that have the most significant impact in the evolution of the God components. In brief, these are the issues with the most classes added and removed. The sample consists of 27 issues in total, which have been analyzed while taking into account two main questions.

6.1 Do developers discuss God components within issues? If yes, what do they discuss?

To answer this question, we have analyzed each issue's comment section in search of discussion related to God components. We have taken into account the fact that the term "God component" does not have an exact definition and therefore can be somewhat freely interpreted.

In the entire sample set, there is no direct mention of "God components", nor is there any word that starts with the noun "God". That being said, the developers indirectly refer to God components by discussing topics such as file size reduction, file count reduction, refactoring hierarchies and removing unnecessary dependencies. The following examples demonstrate issues with the most impact on the evolution of the God components and the discussions of the developers resolving around these components.

Issue Tika-3241 is a task concerning the improvement of the Tika parsers by restructuring and clarifying the hierarchy and parser modules. Besides a brief description, no discussion takes place in the comment section. Regardless of the lack of discussion, this issue has the highest number of classes added and removed, and therefore could still be considered as the most relevant issue within our sample.

Issue Tika-2756 concerns the unnecessary dependency upon the package "common-lang 2.x" and suggests improving parts of code so that they depend on "common-lang 3.x" instead, resulting in the dependency of "common-lang 2.x" being dropped and reducing the software in file size.

Issue TIKA-1343 represents the addition of a new feature in Tika. The new feature focuses on implementing a Tika translator using JoshuaDecoder. The developers managed to add this feature into Tika by using the Joshua Java API. The Tika translator was structured into a new module called tika-translate-joshua, where NetworkTranslator acts as a base class and Joshua acts as a sub-class. The implementation of this feature decreased the file size by removing unnecessary code.

Issue TIKA-1508 is an improvement that suggests reducing the amount of scattered .properties files used for individual parser configurations, creating a sense of uniformity. In the comment section, there is an active discussion on how the issue should be tackled.

The examples above represent some of the issues that had the most significant impact on the evolution of the God component. Even though the developers do not directly mention the God components in the comment section, they indirectly refer to them by discussing topics such as hierarchies, restructuring modules and removing dependencies.

6.2 How do developers re-factor God components?

As mentioned in the previous question, the main refactoring that takes place within Tika focuses on improving the hierarchies, restructuring modules and removing any unnecessary dependencies. Indeed, one might be able to interpret the term 'refactoring' in multiple ways. One might consider refactoring anything, that reduces the complexity of the code. Or, might we agree that refactoring is any commit that **removes** lines of code from the code-base, i.e. God Component. This might not be always the case - we might perfectly fine refactor some code whilst mainly adding lines of code instead of deleting any. The ambiguity of the definition, however, might make it difficult to quantify the metric in a quantitative analysis setup - which is the reason it should be tackled in a **qualitative** way.

We agree on the definition that re-factoring means anything that reduces the complexity of the code. In general, developers in Tika do only **implicitly** refactor God Components. They concern for the maintainability of the code, but are not explicitly involved in reducing the sizes of God Components. It can be found that in a multitude of cases, developers 'restructure' the code, to improve readability and maintainability - ensuring the code-base evolves in a durable manner. In some cases, code is re-factored by moving logic into separate packages and splitting up others inside multiple classes inside a package. Although developers do introduce God Components in several occasions without being aware, it is for sure clear, that the developers care about their code.

7 Conclusion

In summary, we have seen that God Components are indeed present in Apache Tika and have been present for a long time without disappearing. Let us summarize our key insights we gained during this project, as well as reflect what could have gone better in the Future Work section.

7.1 Key insights

God Components exist inside Apache Tika. Using the reverse-engineering tool Designite, we were able to identify God Components for any version of Apache Tika. In order to track the 'evolution' of the God Components over time, we executed Designite on every commit in the Apache Tika code-base, in the main branch present in the repository. After first appearing, most God Components do not disappear - they stay in the system until this date. Note they might stop being a God Component in between, however.

Only a handful of authors are responsible. Although at first sight it seemed the code-base had a pretty high total amount of contributors, it turned out that most developers had only a very minor contribution to the project. Some contributors made only a couple commits, changed a couple lines of code or added a single class or so. The bulk of the project was written by really only a handful of 'fanatic' authors. With only a handful of authors responsible for the code changing, also only a handful of authors are responsible for the classes changing - and therefore the buildup- and refactoring of God Components inside Apache Tika.

Issues of the improvement type are most commonly adding to God Component buildup. Having combined our data coming from various sources, we were able to find out what issues were most responsible for God Component buildup- or refactoring. It turns out that among all issue types, the *Improvement* issue type is most responsible for God Component buildup - in absolute numbers. Relatively, the *New Feature* and *Task* issue types both also seem to 'add' more classes rather than remove them in equivalent ratios; all the just mentioned issue types seem to add about 3 classes for every 2 they remove. An issue type that seems to add and remove about equivalently as much classes is the *Bug* issue type - which rationally speaking makes sense.

7.2 Future work

No project might ever be fully complete and as comprehensive as the authors intend to be: time is always a limiting factor. Although we believe our analysis is quite complete and comprehensive, there are also things we would like to see others do in the future, if the topic were ever addressed again.

Respect branching in delta computation. During the computation of the *chronological classes difference* in Section 4.6.1, a non-ideal assumption is made. This is the assumption that # classes is correctly traceable over time using merely the git commit history on the main branch. In reality, however, code is written simultaneously on different branches, causing commits containing different versions of the code to chronologically follow up on each other. This means that, when we analyse the differential property on the amount of classes on the main branch, they sometimes tend to 'jump' back and forth. This is caused by the fact that code affecting God Components was written simultaneously on different branches and then having its commits intermixed in our dataset after the branches were all merged in the main branch. A solution would be to use the data git has and is able to provide to solve this problem, the information on the **parent** of a git commit. Using this information, a command like `git rev-list --parents` might be used to extract the history whilst respecting the branching in the code-base. Additional processing would have been required, however, to correctly structure the data with respect to the new commit parents information.

Do note, however, that we believe that the impact of the overall results in this project are probably not huge. The 'alternation' issue we mentioned above is indeed a problem, however, it is also the case that if we sum the amount of delta class changes, the alternations sum to a net amount of 0; because they cancel each other out.

Animate the growth of God Components. Another aspiration one might have is that it might be cool to create an animating plot. The animating variable could be in terms of time, whilst keeping the other variables constant. Cool plots that could then be created to visualize the 'evolution' of God Components over time would be, for example, a 'growing' code-base in terms of Lines Of Code or # classes. This might make the interpretation of the results more attractive and fun at the same time.

8 Contributions

Konstantina Gkikopouli

- Wrote the sections "High level architecture", "Key Design Goals"
- Added the quality attributes and their proof, refactored them based on the feedback.
- Analyzed half of the main components in section 3.3
- Analyzed the god components and, in part, answered questions in section 5. Refactored them based on feedback.
- Analyzed issues and wrote parts of section 6, refactored them based on feedback.

Jeroen Overschie

- Wrote introduction
- Wrote data collection code; running Designite, sending job to Peregrine, collecting-, ordering and aggregating the data using Python and Pandas/Numpy.
- Wrote data analysis Jupyter Notebook; load and interpret mined data, write comments and Markdown explanations, plot and fine-tune plots used in report.
- Administered GitHub repo; hosted Notebook on website, wrote README with running instructions.
- Analyzed half of the main components in section 3.3
- Wrote Chapter 4 describing the program functionalities and algorithms.
- Restructured report and in wrote, in part, Chapter 5. Added plots and text around them.

References

[1] ASF Apache Software Foundation. https://tika.apache.org/, Retrieved 23rd of November 2020.

[2] A. B. Burgess and C. A. Mattmann. Automatically classifying and interpreting polar datasets with apache tika. In *Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014)*, pages 863–867, 2014.

[3] C Chedgey, P Hickey, P O'Reilly, and R McNamara. Structure101.

[4] Jukka Zitting Chris Mattmann. *Tika in Action*. Manning, 2011.

[5] J Dresner and KH Borchers. Maintenance, maintainability, and system requirements engineering. Technical report, SAE Technical Paper, 1964.

[6] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Toward a catalogue of architectural bad smells. In Raffaela Mirandola, Ian Gorton, and Christine Hofmeister, editors, *Architectures for Adaptive Software Systems*, pages 146–162, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[7] Wes McKinney et al. pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, 14(9), 2011.

[8] Arthur J Riel. Object-oriented design heuristics (vol. 338). *Reading: Addison-Wesley*, 1996.

[9] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. Designite: A software design quality assessment tool. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, pages 1–4, 2016.

[10] Connie U Smith and Lloyd G Williams. Software performance antipatterns. In *Proceedings of the 2nd international workshop on Software and performance*, pages 127–136, 2000.

A Appendix: Analysis

A.1 Component overview

A.1.1 tika-app

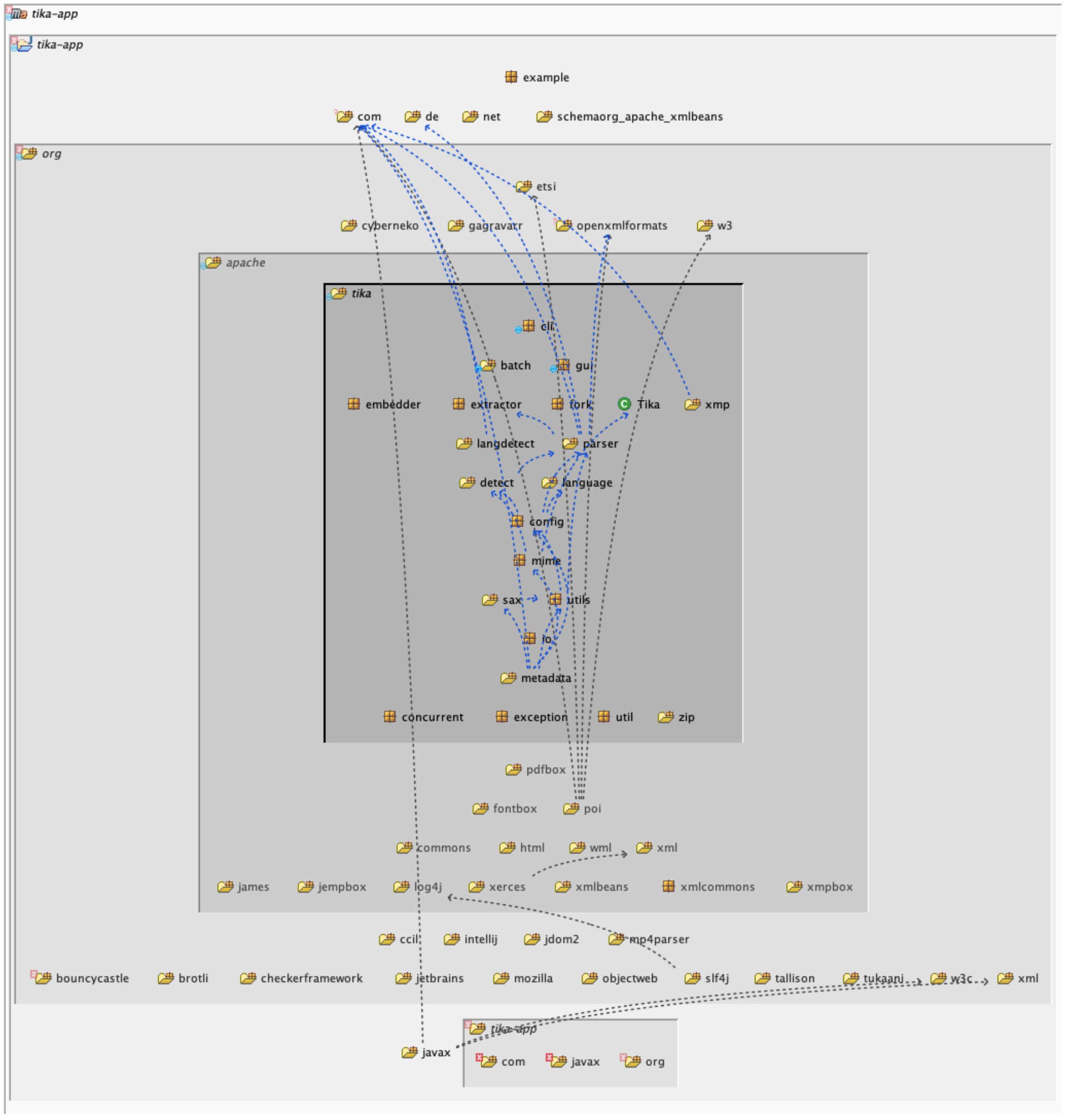


Figure 18: Structure101 overview of tika-app module.

A.1.2 tika-core

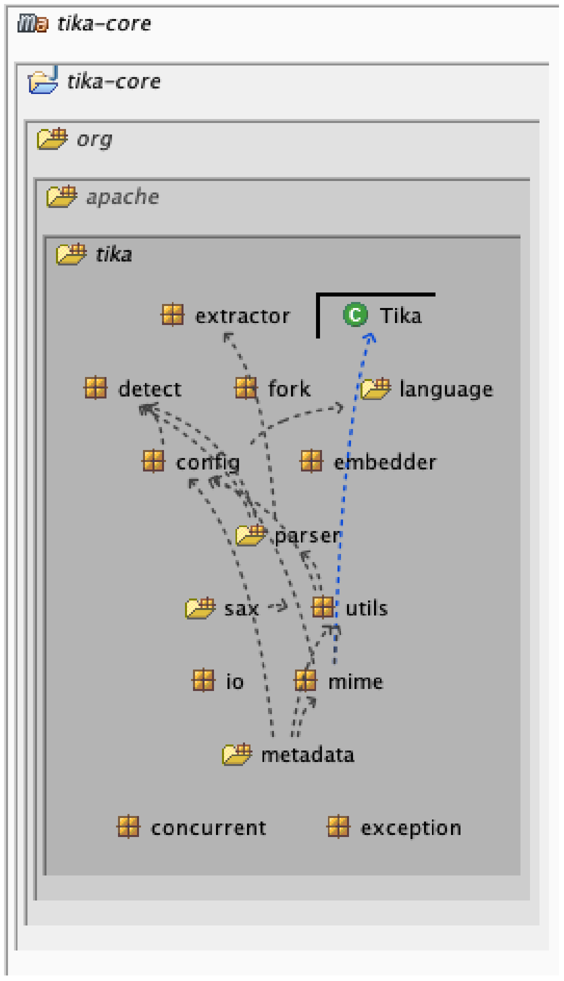


Figure 19: Structure101 overview of tika-core module.