



Commit Analysis Infrastructure (ComAnI)

Guide

September 28, 2022

Christian Kröher
kroehler@sse.uni-hildesheim.de

Implemented in: Java, JavaSE-11
Executable on: Windows and Ubuntu (OpenJDK 11)
Licensed under: Apache License, Version 2.0

Acknowledgment

This work is partially supported by the ITEA3 project REVaMP², funded by the BMBF (German Ministry of Research and Education, <https://www.bmbf.de/>) under grant 01IS16042H. Any opinions expressed herein are solely by the author(s) and not by the BMBF.

A special thanks goes to the developers of KernelHaven [9, 3, 4]: Adam Krafczyk, Sascha El-Sharkawy, Moritz Flöter, Alice Schwarz, Kevin Stahr, Johannes Ude, Manuel Nedde, Malek Boukhari, and Marvin Forstreuter. Their architecture and core concepts significantly inspired the development of ComAnI. In particular, the mechanisms for file-based configuration of the infrastructure and the plug-ins as well as loading and executing individual plug-ins are adopted in this work.

Contents

1	Introduction	5
2	Overview	6
3	User Guide	8
3.1	Installation	8
3.1.1	Downloading ComAnI	8
3.1.2	Satisfying Requirements	8
3.2	Execution	9
4	Developer Guide	13
4.1	Data Model	13
4.2	Commit Extractor Plug-ins	13
4.3	Commit Analyzer Plug-ins	15
	References	19

List of Figures

1	ComAnI overview	6
2	ComAnI data model	13
3	ComAnI <code>AbstractCommitExtractor</code> class	13
4	ComAnI <code>AbstractCommitAnalyzer</code> class	17

Listings

1	ComAnI core configuration parameters	10
2	ComAnI extraction configuration parameters	11
3	ComAnI analysis configuration parameters	12
4	Blueprint of a ComAnI commit extractor main class	16
5	Blueprint of a ComAnI commit analyzer main class	18

1 Introduction

The Commit Analysis Infrastructure (ComAnI) is an open and configurable infrastructure for the extraction and analysis of commits from software repositories. For both tasks, individual plug-ins realize different extraction and analysis capabilities, which rely on the same data model provided by the infrastructure. Hence, any combination of extraction and analysis plug-ins is possible. For example, we could first conduct an analysis for a software hosted in a Git repository [2] and later conduct the same analysis for a different software hosted by SVN [1]¹. Another example is to use the same commit extractor, e.g., supporting the commit extraction from Git repositories, for different analyses. The definition of a particular ComAnI instance consists of a set of configuration parameters saved in a configuration file, which the infrastructure reads at start-up. Hence, there is no implementation effort needed. The infrastructure automatically performs its internal setup, loads and starts the desired plug-ins.

ComAnI represents a large increment of the ComAn toolset [10]. This toolset uses a single commit extraction script and a Java-based implementation of a particular commit analysis [6, 7, 5]. Further, the toolset is designed to be applied to the Linux kernel [8] and its Git repository [11]. This design of ComAn restricts its applicability to other software and repository types. While it is not completely impossible to adapt it to other inputs, this adaptation requires mayor implementation effort. Hence, we decided to create a complete infrastructure, which realizes a flexible and highly configurable ecosystem for conducting a variety of analyses by means of plug-ins for commit extraction and their analysis.

This guide consists of three parts. The first part in Section 2 introduces ComAnI in more detail and describes the concepts realizing the core features of the infrastructure. Section 3 represents the second part, which focuses on the end user of ComAnI. We describe how to download, install and execute the core infrastructure as well as the available plug-ins. As part of the execution, we also discuss the configuration parameters and the definition of particular ComAnI instances. The third part of this guide focuses on the developers. In Section 4, we discuss the development of new extraction and analysis plug-ins by examples.

¹Assuming that the analysis is able to cope with the artifacts and their technologies of the new software under analysis.

2 Overview

ComAnI is designed to support the extraction of commits from different version control systems and various analyses of those extracted commits in any combination. For this purpose, it offers an open plug-in infrastructure implemented in Java. The core components of this infrastructure are the commit extractor, the internal data model, the commit analyzer, and the configuration file as illustrated in Figure 1. In this section, we will describe these components in detail.

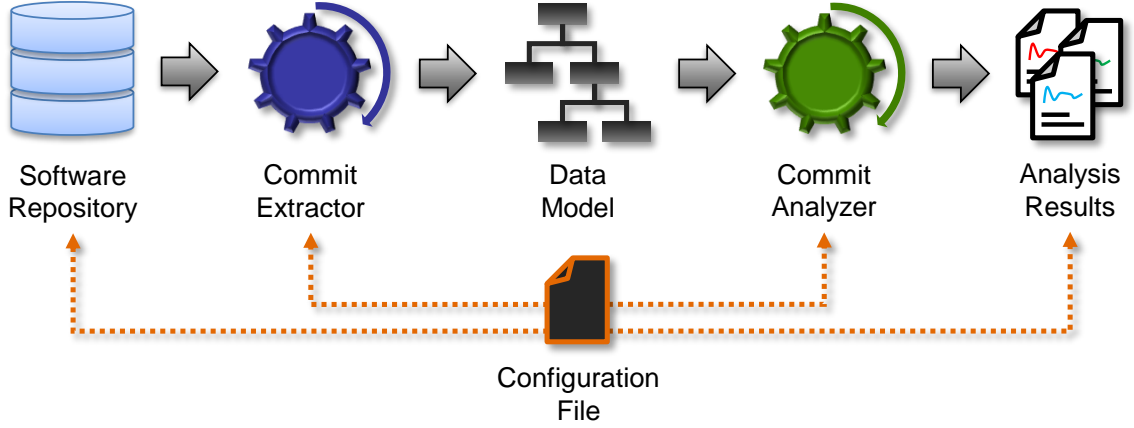


Figure 1: ComAnI overview

A **commit extractor** is a ComAnI plug-in, which is responsible for the extraction of commits and their provision for the commit analysis as elements of the internal data. A commit extractor typically supports three extraction variants² depending on the given sources from which the commits shall be extracted:

- *Full repository extraction*: this variant forces the extraction of all commits of a software repository. This requires the definition of the location of the target repository as part of the configuration file.
- *Partial repository extraction*: instead of extracting all commits of a software repository, this variant allows the extraction of a predefined set of commits. Besides the location of the target repository, this requires the specification of an additional file, which contains a list of unique commit numbers (or hashes). Each line of this commit list file must contain exactly one commit number. Further, the author of the commit list file must ensure that the commit numbers specify commits of the target repository.
- *Single commit extraction*: the third variant offers an interactive mode, in which the content of a single commit can be passed on the command line as an input. It must be terminated by a last line only containing the string `!q!`. If this line is missing, the extraction will fail. Further, all lines after `!q!` will be ignored. However, the exact format of that string representing a single commit depends on the particular commit extractor. Please read the respective part of the documentation of the commit extractor in use for this information (typically defined in the `README.md` file).

The available extractors are introduced in Section 3.1. Section 3.2 describes their definition for a particular ComAnI instance and the usage of the different extraction variants above. Further, Section 4.2 explains the development of new extractor plug-ins for custom commit extraction capabilities.

²While the respective methods need to be implemented by each commit extractor, developers are free to realize the required algorithms. Hence, we cannot guarantee that all extractors support all extraction variants. We recommend reading the description of the desired commit extractor for more information.

The internal **data model** represents the conceptual interface between commit extractors and commit analyzers. It offers two main elements for representing commits: the **Commit** itself, which provides information, like its id (the commit number) or date, and the **ChangedArtifact** for storing the information about the artifacts changed by a specific commit. Hence, each commit typically contains a list of changed artifacts, which in turn contain information about their name and location as well as their content including the changed lines. A commit extractor creates instances of these elements based on the extracted commits from a target repository or the content of a commit passed as command line input. Section 4.1 provides further details about the internal data model and its elements.

The elements of the internal data model are input to a **commit analyzer**, which is a ComAnI plug-in similar to a commit extractor. Depending on the core algorithm of the respective analysis, a commit analyzer may either wait until all commits are available or directly start processing at the time a commit is available. The infrastructure neither imposes any restriction on the way of processing nor on the analysis results. Hence, each commit analyzer has full control over its result creation. The only input it receives is an output directory in which the results can be stored. While the available analyzers are introduced in Section 3.1, their definition for and usage in a ComAnI instance are described in Section 3.2. Section 4.3 explains the development of this type of plug-ins in detail.

The **configuration file** in the lower part of Figure 1 defines a particular setup of the commit extraction and analysis and, hence, a specific instance of ComAnI. It consists of a set of configuration parameters for preparing the infrastructure (input and output locations, etc.) as well as defining the desired commit extractor and analyzer plug-ins. The infrastructure reads these parameters to configure ComAnI prior to its actual execution. Section 3.2 introduces the available configuration parameters and their definition for a particular ComAnI instance.

3 User Guide

Users apply their specific instance of ComAnI to analyze particular properties of a software repository and its commits. Hence, we describe how to install and execute such an instance in this section. Further, we explain how to satisfy requirements for a successful execution as well as the main configuration parameters for defining a ComAnI instance.

3.1 Installation

This section describes the installation of the main infrastructure as well as how to satisfy its requirements. We provide this description for both operating systems currently used to execute ComAnI: Ubuntu and Windows. However, the first part in Section 3.1.1 is operating system independent, while Section 3.1.2 contains descriptions for both operating systems.

3.1.1 Downloading ComAnI

We recommend downloading the *ComAnI.zip* archive, which already contains the main infrastructure, the guide, and the plug-ins developed by the SSE group. This requires the following steps, which are operating system independent:

1. Go to <https://github.com/CommitAnalysisInfrastructure/ComAnI>
2. Navigate to the release directory
3. Click on the *ComAnI.zip* file
4. Click the Download button and save the archive
5. Go to the download directory and extract all files in the ComAnI.zip archive to your favorite directory

The directory to which the ComAnI-plug-ins are extracted in step 5 above is used for the definition of the configuration file in Section 3.2.

3.1.2 Satisfying Requirements

The execution of ComAnI requires the installation of Java 11 or higher (or equivalents, like OpenJDK). The required steps for this installation are operating system dependent. Hence, we first describe the necessary steps for Ubuntu and then for Windows below.

On Ubuntu, the following steps have to be performed:

1. Open a Terminal
2. Update the apt package index: `sudo apt update`
3. Install OpenJDK:
 - (a) Either the default, if it is OpenJDK 11 or higher: `sudo apt install default-jdk`
 - (b) Or explicitly OpenJDK 11: `sudo apt install openjdk-11-jdk`
4. Check if the installation was successful by printing the installed Java version: `java -version`

On Windows, the following steps have to be performed:

1. Distribution-dependent installation:
 - (a) For Oracle Java:

- i. Go to <https://www.java.com/en/download/> (Oracle Java)
 - ii. Download the Java Runtime Environment (JRE) for your system
 - iii. Execute the installer and follow its instructions
 - (b) For OpenJDK:
 - i. Go to <https://jdk.java.net/java-se-ri/11>
 - ii. Download the RI Binary (build 11+28) Windows/x64 Java Development Kit
 - iii. Extract the content of that archive to your favorite location
2. Check if the installation was successful by printing the installed Java version:
- (a) Open the Command Prompt: Hit the Windows-Key or click Start, type `cmd` and hit Enter
 - (b) In the Command Prompt: `java -version`

3.2 Execution

The execution of ComAnI requires the following preparation steps:

1. Clone or checkout the software repository you want to analyze to your favorite directory
2. Create a ComAnI configuration file, e.g. *my-instance.properties*

For the second step, the infrastructure offers the parameters described in Listing 1 to Listing 3.

The actual execution takes place via the following steps:

1. Open a Terminal (Ubuntu) or Command Prompt (Windows)
2. Navigate to the ComAnI directory, e.g., the one used in step 5 in Section 3.1.1
3. Start the infrastructure with the desired configuration file:
`java -jar ComAnI my-instance.properties`

Please note that there may be plug-in-specific configuration parameters, which are not covered by the following listings. We recommend reading the respective Github pages to find out more about such parameters and their usage.

```

1 # The path to the directory, which contains the ComAnI plug-ins,
2 # like the available extractors and analyzers.
3 #
4 # Type: mandatory
5 # Default value: none
6 # Related parameters: none
7 core.plugins_dir = <Path>
8
9 # The identifier of the version control system (VCS), which the
10 # repository as the input for commit extraction relies on.
11 # Commit extractors and analyzers need to support the VCS. See
12 # the respective documentations of the desired plug-ins.
13 #
14 # Type: mandatory
15 # Default value: none
16 # Related parameters: none
17 core.version_control_system = <VCS_Id>
18
19 # The number defining a particular log-level and, hence, the
20 # amount of information the infrastructure as well as the plug-ins
21 # provide at runtime.
22 # Valid values are:
23 #     0 – SILENT: No information is provided and, hence, there will
24 #                be no message at all except for initial setup
25 #                errors
26 #     1 – STANDARD: Basic information, warnings, and errors are
27 #                provided
28 #     2 – DEBUG: Similar to STANDARD, but additional debug
29 #                information is provided
30 #
31 # Type: optional
32 # Default value: 1
33 # Related parameters: none
34 core.log_level = <0|1|2>
35
36 # The maximum number of commits, which the commit queue manages
37 # simultaneously for their transfer from the extractor to the
38 # analyzer. If this number is reached, the queue blocks the
39 # extractor until the analyzer removes the next commit from the
40 # queue.
41 #
42 # Type: optional
43 # Default value: 10
44 # Related parameters: none
45 core.commit_queue.max_elements = <1|2|3|...|100000>

```

Listing 1: ComAnI core configuration parameters

```

1 # The fully qualified main class name of the commit extractor
2 # to use in the particular ComAnI instance. Although being
3 # mandatory, the infrastructure will ignore this parameter, if
4 # reuse is enabled.
5 #
6 # Type: mandatory
7 # Default value: none
8 # Related parameters: none
9 extraction.extractor = <Extractor>
10
11 # The path to the directory denoting the root of a software
12 # repository from which the commit extractor will extract the
13 # commits. Although being mandatory, extractors will ignore
14 # this parameter in interactive mode.
15 #
16 # Type: mandatory
17 # Default value: none
18 # Related parameters: none
19 extraction.input = <Path>
20
21 # The path to and name of the file containing a list of commit
22 # numbers. Extractors will try to extract the corresponding
23 # commits from the specified repository exclusively.
24 #
25 # Type: optional
26 # Default value: none
27 # Related parameters: none
28 extraction.commit_list = <Path>
29
30 # The path to the directory for saving extracted commits.
31 # Defining this parameter enables the caching feature for the
32 # extraction, which allows saving extracted commits as individual
33 # files and reuse them in future analyses, while the current
34 # analysis processes the extracted commits as usual. This avoids
35 # repeating the extraction of the same commits for future analyses.
36 #
37 # IMPORTANT: the infrastructure deletes the content of this
38 # directory, if it is not empty.
39 #
40 # Type: optional
41 # Default value: none
42 # Related parameters: extraction.reuse
43 extraction.cache = <Path>
44
45 # The path to the directory containing cached commits. Defining
46 # this parameter enables the caching feature for the extraction,
47 # which leads to a reuse of previously extracted commits instead of
48 # executing the defined extractor. This avoids repeating the
49 # extraction of the same commits for future analyses.
50 #
51 # IMPORTANT: if caching and reusing is defined at the same time,
52 # caching is performed and the analysis uses extracted commits.
53 #
54 # Type: optional
55 # Default value: none
56 # Related parameters: none
57 extraction.reuse = <Path>

```

Listing 2: ComAnI extraction configuration parameters

```

1 # The fully qualified main class name of the commit analyzer
2 # to use in the particular ComAnI instance.
3 #
4 # Type: mandatory
5 # Default value: none
6 # Related parameters: none
7 analysis.analyzer = <Analyzer>
8
9 # The path to the directory for saving the analysis results.
10 # Each analysis will create its own sub-directory in this
11 # directory named by the name of the analyzer and a timestamp
12 # to avoid unintended overriding of previous results.
13 #
14 # Type: mandatory
15 # Default value: none
16 # Related parameters: none
17 analysis.output = <Path>

```

Listing 3: ComAnI analysis configuration parameters

4 Developer Guide

Developers create their own plug-ins for the main infrastructure. Hence, we describe the necessary information about such development in this section. However, we focus on developing extractor plug-ins as they must implement different methods and, hence, capabilities. The realization of an analysis plug-in is rather straightforward and shares methods with the extractors (explained there in detail). Hence, we only outline a new analysis plug-in by a blueprint of the main commit analyzer class. For both extractor and analysis plug-ins to be used with the main infrastructure, the respective projects must be exported as a simple Java archive file.

4.1 Data Model



Figure 2: ComAnI data model

4.2 Commit Extractor Plug-ins

A commit extractor plug-in is responsible for extracting commit information from a software repository and providing this information for an analysis. Therefore it has to create instances of the **Commit** and **ChangedArtifact** classes described in Section 4.1 and add these instances to the internal **CommitQueue**. This queue represents the actual connection between commit extractors and analyzers. It is accessible through an attribute of the **AbstractCommitExtractor** class, which each commit extractor has to extend. Figure 3 presents this abstract class as well as the extractor-specific commit queue interface.

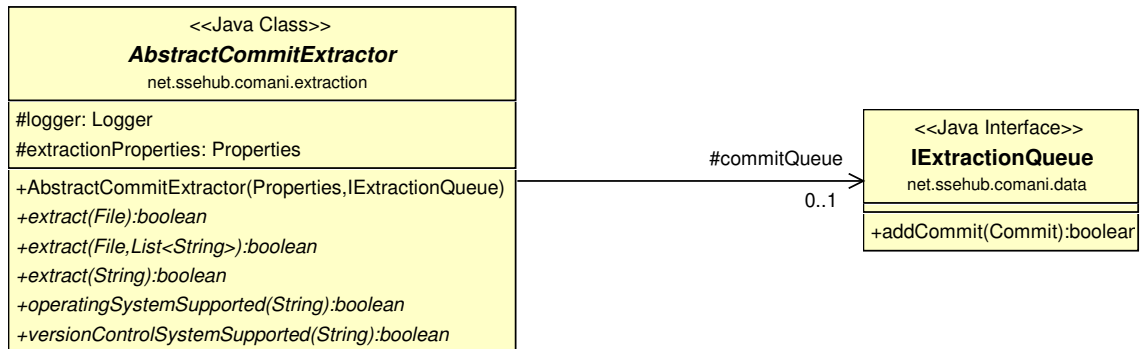


Figure 3: ComAnI **AbstractCommitExtractor** class

Each commit extractor inherits three attributes: the infrastructure-wide logger, the extraction properties, and the commit queue as shown in Figure 3. The **logger** provides multiple methods for logging general information about the extraction process, warning,

error, and debug messages. The amount of information actually shown, e.g., on the command line, depends on the defined log-level in the configuration file (cf. Section 3.2). The `extractionProperties` include all configuration parameters, which start with the prefix “`extraction.`”, a property providing the name of the operating system on which the extractor currently runs, and a property for the version control system as specified by the respective configuration parameter in the configuration file (cf. Section 3.2). The `commitQueue` enables the transfer of extracted commits to an analysis. It only provides a single method, which accepts a single `Commit` instance as a parameter. Hence, the extraction algorithms have to call this method for each extracted commit individually.

Figure 3 also shows that a commit extractor has to implement a constructor, which accepts a properties and a particular extraction queue instance as parameters, as well as a set of methods for extracting commits and checking whether it is executable in the current environment. Listing 4 introduces a blueprint of a commit extractor, which implements all these required elements.

This blueprint represents a starting point for each new extractor by implementing the necessary algorithms as follows:

1. *Constructor*: creates a new instance of the commit extractor and, hence, has to call its parent class’ constructor by passing the constructor parameters of the new extractor. Further actions for setting up the particular commit extractor can be performed here as well, which may also throw `ExtractionSetupExceptions`, if the setup fails. Listing 4 shows the constructor in Lines 16 to 22 including the usage of the `logger`, which informs the user about its creation (Line 20).
2. *Extraction methods*: realize the three extraction variants as introduced in Section 2. Each method in the Lines 25 to 41 returns a Boolean value indicating whether the particular extraction variant was successful (`true`) or not (`false`). In the latter case, the user is automatically informed about an extraction error indicating that either there are no analysis results or the results may potentially be incorrect. The individual methods have the following purpose:
 - (a) `extract (File repository)`: extracts all commits of the given software repository. The `repository` parameter identifies the directory specified as input (`extraction.input`) in the configuration file (cf. Section 3.2), which is typically the root directory of a software repository. The particular way of interacting with the supported type of repository depends on the commands and capabilities provided by the version control system.
 - (b) `extract (File repository, List<String> commitList)`: extracts only those commits of the given software repository, which are part of the given commit list. While the `repository` parameter provides the same information as for the method above, the `commitList` parameter contains a set of commit numbers (or hashes), which enable to extraction of the respective commits. However, this method is only called if a commit list is defined via the corresponding configuration parameter in the configuration file.
 - (c) `extract (String commit)`: transforms the given information representing the content of a particular commit into a commit of the internal data model. This method is only called in the interactive mode of ComAnI, in which the given string is passed directly as a command line argument.
3. *Support check methods*: realize the opportunity to restrict the application of a commit extractor to a particular operating system and version control system. In particular, this is important, if, for example, an extractor relies on a third-party library, which is only available for Windows, or if the extractor cannot process other commits than those

of a Git repository. A missing support yields a `ExtractionSetupException` during the creation of an instance of the extractor by the infrastructure, which terminates the entire tool. Each method in Lines 44 to 54 returns a Boolean value indicating whether the extractor supports the respective system (`true`) or not (`false`):

- (a) `operatingSystemSupported(String os)`: checks whether the extractor supports the given operating system. The `os` parameter provides the operating system in the format of `System.getProperty("os.name")`³.
- (b) `versionControlSystemSupported(String vcs)`: checks whether the extractor supports the given version control system. The `vcs` parameter provides the version control system as defined by the `core.version_control_system` configuration parameter in the configuration file.

4.3 Commit Analyzer Plug-ins

³<https://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html>

```

1 package core;
2
3 import java.io.File;
4 import java.util.List;
5 import java.util.Properties;
6
7 import net.ssehub.comani.core.Logger;
8 import net.ssehub.comani.data.IExtractionQueue;
9 import net.ssehub.comani.extraction.AbstractCommitExtractor;
10 import net.ssehub.comani.extraction.ExtractionSetupException;
11
12 public class CommitExtractor extends AbstractCommitExtractor {
13
14     private static final String ID = "MyCommitExtractor";
15
16     public CommitExtractor(Properties extractionProperties,
17         IExtractionQueue commitQueue)
18         throws ExtractionSetupException {
19         super(extractionProperties, commitQueue);
20         this.logger.log(ID, "Created", null, Logger.MessageType.INFO);
21         // TODO Further setup actions go here
22     }
23
24     @Override
25     public boolean extract(File repository) {
26         // TODO Extraction of all commits from given repository
27         return false;
28     }
29
30     @Override
31     public boolean extract(File repository, List<String> commitList) {
32         /* TODO Extraction of all commits of given commit list from
33          * given repository */
34         return false;
35     }
36
37     @Override
38     public boolean extract(String commit) {
39         // TODO Extraction of given commit (convert to data model)
40         return false;
41     }
42
43     @Override
44     public boolean operatingSystemSupported(String os) {
45         // TODO Check if extractor supports given operating system
46         return false;
47     }
48
49     @Override
50     public boolean versionControlSystemSupported(String vcs) {
51         /* TODO Check if extractor supports given version control
52          * system */
53         return false;
54     }
55 }
56

```

Listing 4: Blueprint of a ComAnI commit extractor main class

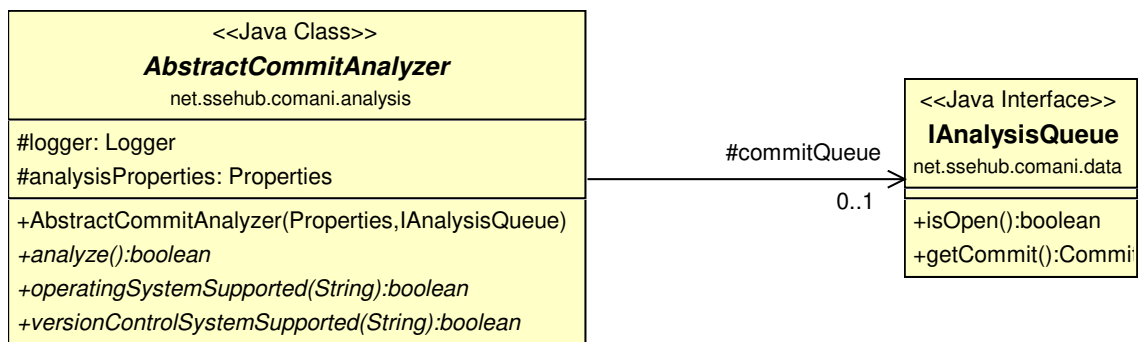


Figure 4: ComAnI AbstractCommitAnalyzer class

```

1 package core;
2
3 import java.util.Properties;
4
5 import net.ssehub.comani.analysis.AbstractCommitAnalyzer;
6 import net.ssehub.comani.analysis.AnalysisSetupException;
7 import net.ssehub.comani.core.Logger;
8 import net.ssehub.comani.data.Commit;
9 import net.ssehub.comani.data.IAnalysisQueue;
10
11 public class CommitAnalyzer extends AbstractCommitAnalyzer {
12
13     private static final String ID = "MyCommitAnalyzer";
14
15     public CommitAnalyzer(Properties analysisProperties,
16         IAnalysisQueue commitQueue)
17         throws AnalysisSetupException {
18         super(analysisProperties, commitQueue);
19         this.logger.log(ID, "Created", null, Logger.MessageType.INFO);
20         // TODO Further setup actions go here
21     }
22
23     @Override
24     public boolean analyze() {
25         while (this.commitQueue.isOpen()) {
26             Commit commit = this.commitQueue.getCommit();
27             if (commit != null) {
28                 // TODO Analyze commit
29             }
30         }
31         return false;
32     }
33
34     @Override
35     public boolean operatingSystemSupported(String os) {
36         // TODO Check if analyzer supports given operating system
37         return false;
38     }
39
40     @Override
41     public boolean versionControlSystemSupported(String vcs) {
42         /* TODO Check if analyzer supports given version control
43         * system */
44         return false;
45     }
46 }
47

```

Listing 5: Blueprint of a ComAnI commit analyzer main class

References

- [1] Apache Software Foundation. Apache subversion. <https://subversion.apache.org/>, 2018. Accessed 2018/10/11.
- [2] Git. Git version control system. <https://git-scm.com/>, 2018. Accessed 2018/10/11.
- [3] C. Kröher, S. El-Sharkawy, and K. Schmid. KernelHaven - an experimentation workbench for analyzing software product lines. In *40th International Conference on Software Engineering: Companion Proceedings*, pages 73–76, New York, NY, USA, 2018. ACM.
- [4] C. Kröher, S. El-Sharkawy, and K. Schmid. KernelHaven - an open infrastructure for product line analysis. In *22nd International Systems and Software Product Line Conference*, volume 2, pages 5–10, New York, NY, USA, 2018. ACM.
- [5] C. Kröher, L. Gerling, and K. Schmid. Identifying the intensity of variability changes in software product line evolution. In *22nd International Systems and Software Product Line Conference*, volume 1, pages 54–64, New York, NY, USA, 2018. ACM.
- [6] C. Kröher and K. Schmid. A commit-based analysis of software product line evolution: Two case studies. Technical Report SSE 2/17/E, University of Hildesheim, 2017.
- [7] C. Kröher and K. Schmid. Towards a better understanding of software product line evolution. In *Softwaretechnik-Trends*, volume 37:2, pages 40–41, Berlin, Germany, 2017. Gesellschaft für Informatik e.V., Fachgruppe PARS.
- [8] Linux Kernel Organization, Inc. The Linux kernel archives. <https://www.kernel.org/>, 2018. Accessed 2018/10/15.
- [9] Stiftung Unviversity of Hildesheim - Software Systems Engineering. KernelHaven. <https://github.com/KernelHaven>, 2018. Accessed 2018/10/12.
- [10] Stiftung Unviversity of Hildesheim - Software Systems Engineering. Variability-centric commit extraction and analysis. <https://github.com/SSE-LinuxAnalysis/ComAn>, 2018. Accessed 2018/10/15.
- [11] L. Torvalds. Linux kernel source tree. <https://github.com/torvalds/linux>, 2018. Accessed 2018/10/15.