Analyzing Code Comments

Pooja Rani
Research Assistant
PhD student

Software Composition Group
University of Bern, Switzerland
Roadmap

- Importance of code comments
- Code comment types
- What is a good comment?
- Challenges
- Tool support
- Various approaches to analyze
We use different tools and techniques to understand code.
Understanding code...

Not So Happy Developers

Happy Developers

Absence of Comments in the Code

Comments in the Code
In GT, try to understand the class “BrLook”, “BIElement” without class comments.
I define how widgets look. In addition to the BrViewModel I listen to UI events and update decoration (non meaningful) elements of the widgets.

Looks install themselves on Brick graphical widgets, and are able to modify the Bloc element tree of the widget. As such, they are very powerful, but should not be used as a hammer for all situations, in particular:

- They should never affect the API of the widget.
- They should not be used to add or remove content in the widget. Element composition is a better solution for this.
Code comments

- Code comprehension tasks
- Code maintenance tasks
- To understand a new domain
You do not need comments if you write clean code.
You do not need comments if you write clean code?
Code can describe how but it cannot explain why.
Code comment types

- Documentation (/** … */)
  (also used for packages / classes / methods)

- Block comments (/* … */)

- Inline comments (//….)
Programming languages follow different syntaxes for comments.

However, most languages support a distinct delimiter for comments.
Class comment example:

```java
/**
 * A class representing a window on the screen.
 * For example:
 * <pre>
 * Window win = new Window(parent);
 * win.show();
 * </pre>
 * 
 * @author Sami Shaio
 * @version 1.13, 06/08/06
 * @see java.awt.BaseWindow
 * @see java.awt.Button
 */

class Window extends BaseWindow {
    ...
}
```

Java class comment
class ExampleClass(object):
    """
The summary line for a class docstring should fit on one line.

If the class has public attributes, they may be documented here in an `Attributes` section and follow the same formatting as a function's `Args` section. Alternatively, attributes may be documented inline with the attribute's declaration (see `__init__` method below).

Properties created with the `@property` decorator should be documented in the property's getter method.

Attributes
___________

attr1 : str
    Description of `attr1`.
attr2 : :obj:`int`, optional
    Description of `attr2`.

"""
I collect Spotter search pragmas. Each pragma is a Spotter extension for a given `object`. By default, I look for `gtSearch` pragmas. It can be changed by `pragmaName`.

The `spotterStep` can decide whether or not an extension is enabled. It can also configure each extension, e.g., override any property.
Method comment example:

```java
/**
 * Returns the character at the specified index. An index
 * ranges from `<code>0</code>` to `<code>length() - 1</code>`
 *
 * @param index the index
 * @return the character
 * @exception StringIndexOutOfBoundsException if an index
 *      is &lt; 0 or &gt;= length()
 */

public char charAt(int index) {
    ...
```

Java method comment:

```java
flush: aFlushBlock

"Process all currently available items, passing each item to a flush block. If there is another process, which currently fetching items from queue, or queue is empty, return immediately"

| item |

| object |

| object := item object. |

| object := dummy ifFalse: [ |

| aFlushBlock value: object ] ifCurtailed: [ |

| item object: dummy. |

| dummy next: item next ]. |

| item object: dummy. |

| item isCircular ifTrue: [ |

| "this was the last one" |

| dummy next: item. |

| self signalNoMoreItems=. |

| ^ self |

| ]. |

| item := item next. |

] repeat.
```
What is a good comment?

/*
 * Dear Maintainer
 * Once you are done trying to ‘optimize’ this routine,
 * and you have realized what a terrible mistake that was,
 * please increment the following counter as a warning
 * to the next guy.
 *
 * total_hours_wasted_here = 73
 * /

// When I wrote this, only God and I understood what I was doing

// Now, God only knows

#This is brilliant
#Thanks. It’s nap time.
What is a good comment?

/*
 * Dear Maintainer
 * Once you are done trying to ‘optimize’ this routine,
 * and you have realized what a terrible mistake that was,
 * please increment the following counter as a warning
 * to the next guy.
 * total_hours_wasted_here = 73
 *
// When I wrote this, only God and I understood what I was doing

// Now, God only knows

#This is brilliant
#Thanks. It’s nap time.
What is a good comment?

- Helps other developers in working with your code
- Describes why, and not how
- Reveals intent, limitation, assumptions, design decisions
- Justifies the violation of a programming style
What is a good comment?

// format matched kk:mm:ss EEE, MMM dd, yyy
Pattern timePattern = Pattern.compile("\d*:\d*:\d* \w*, \w*, \d*, \d*");

Note that to encode a String as Base64, you first have to encode the characters as bytes using character encoder.

It makes sense to use me if scalable element has fixed or matching parent horizontal size but fits content vertically.
Coding style guidelines

- Agreed guidelines to express the information
- To write consistent & informative comments
Guideline examples: Oracle

- Use blank lines after summary line.
- Use 3rd person instead of 2nd person.
- Write a one-line summary of the class.
Coding style guidelines

- **Java**: Oracle, Apache, Google
- **Python**: Pep, Google, Numpy
- **Smalltalk**: Smalltalk style guide, comment template
- **Ruby**: RubyStyle

Oracle: https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html
Numpy: https://numpydoc.readthedocs.io/en/latest/format.html#docstring-standard
Smalltalk: http://sdmeta.gforge.inria.fr/FreeBooks/WithStyle/SmalltalkWithStyle.pdf
Please comment me using the following template inspired by Class Responsibility Collaborator (CRC) design:

For the Class part: State a one line summary. For example, "I represent a paragraph of text".

For the Responsibility part: Three sentences about my main responsibilities - what I do, what I know.

For the Collaborators Part: State my main collaborators and one line about how I interact with them.

Public API and Key Messages
- message one
- message two
- (for bonus points) how to create instances.

One simple example is simply gorgeous.

Internal Representation and Key Implementation Points.

Instance Variables
environmentDictionaries: <Object>

Implementation Points

Pharo class comment template
Challenges

- Multiple conventions
- Personal style
- Incomplete comments
- Outdated comments
- Inconsistent writing style
- Complex comments

Impact overall quality of comments
We use different tools and techniques to analyze code quality.
Code analysis tools

- Syntax
- Semantics
- Style
But what about comments?
Comment analysis tools

- Syntax
- Semantics
- Style
Comment analysis tools

- Syntax
- Semantics
- Style

- Javadoc, pydoc
- IDE
- Data flow Analysis
- Control flow analysis
- Testing tools
- Runtime tools
- Coding style checkers
- Static analysis tools
- Checkstyle, Pylint
Documentation tools

- Check syntax of comments
- Do not check the content
Coding style checkers

- **Java**: Checkstyle, PMD
- **Python**: pylint, pycodestyle
- **Smalltalk**: No linter
- **Ruby**: RuboCop
Coding style checkers

- Detect presence/absence of comments
- Check whether comments follow style guidelines
- Limited to selected metrics (code/comment ratio)
Comment Content Analysis
Information types in Java code comments

Luca Pascarella  
Delft University of Technology  
L.Pascarella@tudelft.nl  

Alberto Bacchelli  
Delft University of Technology  
A.Bacchelli@tudelft.nl

Abstract—Code comments are a key software component containing information about the underlying implementation. Several studies have shown that code comments enhance the readability of the code. Nevertheless, not all the comments have the same goal and target audience. In this paper, we investigate how six diverse Java OSS projects use code comments, with the aim of understanding their purpose. Through our analysis, we produce a taxonomy of source code comments; subsequently, we investigate how often each category occurs in manually classified comments. In addition, we conduct an initial evaluation on how to automatically classify code comments at line level into our taxonomy using machine learning; initial results are promising and suggest that an accurate classification is within reach.

I. INTRODUCTION

While writing and reading source code, software engineers routinely introduce code comments [6]. Several researchers investigated the usefulness of these comments, showing that thoroughly commented code is more readable and maintainable. For example, Woodfield et al. conducted one of the first experiments demonstrating that code comments improve program readability [35]; Tenny et al. confirmed these results with more experiments [31, 32]. Hatzimichail et al. investigated the economical maintenance of large software products showing that comments are crucial for maintenance [12]. Jiang et al. found that comments are misaligned to the annotated functions confuse authors of future code changes [15]. Overall, given these results, having abundant comments in the source code is a recognized good practice [4]. Accordingly, researchers proposed to evaluate code quality with a new metric based on code comments ratio [21, 9].

Nevertheless, not all the comments are the same. This is evident, for example, by glancing through the comments in a source code file from the Java Apache Hadoop Framework [1]. In fact, we see that some comments target end-user programmers (e.g., JavaDoc), while others target internal developers (e.g., inline comments, example with comment).

Hasuuri et al. [11] and Steffl et al. [28] presented the earliest and most significant results in comments’ classification. Hasuuri et al. investigated developers’ commenting habits, focusing on the position of comments with respect to source code and proposing an initial taxonomy that includes four high-level categories [11]. Steffl et al. proposed a semi-automated approach for the quantitative and qualitative evaluation of comment quality, based on classifying comments in seven high-level categories [28]. In spite of the innovative techniques they proposed to both understanding developers’ commenting habits and assessing comments’ quality, the classification of comments was not in their primary focus.

In this paper, we focus on increasing our empirical understanding of the types of comments that developers write in source code files. This is a key step to guide future research on this topic. Moreover, this increased understanding has the potential to (1) improve current quality analysis approaches that are restricted to the comment ratio metric only [21, 9] and (2) strengthen the reliability of other existing approaches that use source code comments as input (e.g., [30, 23]).

To this aim, we conducted an in-depth analysis of the comments in the source code files of six OSS systems in Java. We set up our study as an exploratory investigation. We started without hypotheses regarding the content of source code comments, with the aim of discovering these purposes and roles, their format, and their frequency. To this end, we (1) conducted three iterative comment analysis sessions (involving four researchers) over 50 source files including about 250 comment blocks to define an initial taxonomy of code comments, (2) validated the taxonomy externally with 3 developers, (3) inspected 2,000 source code files and manually classified (using a new application we devised for this purpose) over 15,000 comment blocks comprising more than 28,000 lines, and (4) used the resulting dataset to evaluate how effectively comments can be automatically classified.
Information types in Python code comments

Classifying Python Code Comments Based on Supervised Learning

Jingyi Zhang\textsuperscript{1}, Lei Xu\textsuperscript{2(ES)}, and Yanhui Li\textsuperscript{2}

\textsuperscript{1} School of Management and Engineering, Nanjing University, Nanjing, Jiangsu, China
jyzhangchn@outlook.com
\textsuperscript{2} Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu, China
\{xlei, yanhui1\}@nju.edu.cn

Abstract. Code comments can provide a great data source for understanding programmer’s needs and underlying implementation. Previous work has illustrated that code comments enhance the reliability and maintainability of the code, and engineers use them to interpret their code as well as help other developers understand the code intention better. In this paper, we studied comments from 7 python open source projects and contrived a taxonomy through an iterative process. To clarify comments characteristics, we deploy an effective and automated approach using supervised learning algorithms to classify code comments according to their different intentions. With our study, we find that there does exist a pattern across different python projects: Summary covers about 75\% of comments. Finally, we conduct an evaluation on the behaviors of two different supervised learning classifiers and find that Decision Tree classifier is more effective on accuracy and runtime than Naive Bayes classifier in our research.

Keywords: Code comments classification · Supervised learning · Python
Information types in Pharo code comments

What do class comments tell us? An investigation of comment evolution and practices in Pharo Smalltalk

Pooja Rani · Sebastiano Panichella · Manuel Leuenberger · Mohammad Ghafari · Oscar Nierstrasz

Received: date / Accepted: date

Abstract Previous studies have characterized code comments in various programming languages, and have shown how a high quality of code comments is crucial to support program comprehension activities and to improve the effectiveness of maintenance tasks. However, very few studies have focused on the analysis of the information embedded in code comments. None of them has compared developer practices to write comments following the standard guidelines or analyzed these characteristics in the Pharo Smalltalk environment.

These class commenting practices have their origins in Smalltalk-80, going back 40 years. Smalltalk traditionally separates class comments from source code, and offers a brief template for entering a comment for newly-created classes. These templates have evolved over the years, particularly in the Pharo environment. This paper reports the first empirical study investigating commenting practices in Pharo Smalltalk. As a first step, we analyze class comment evolution over seven Pharo versions. Then, we quantitatively and qualitatively analyze class comments of the most recent version of Pharo, to investigate the information types of Pharo comments. Finally, we study the adherence of developer commenting practices to the class template over Pharo versions.

The results of this study show that there is a rapid increase in class comments in the initial three Pharo versions, while in subsequent versions developers added comments to both new and old classes, thus maintaining a similar ratio. In addition, the analysis of the semantics of the comments from the latest Pharo version suggests that 23 information types are typically embedded in class comments by developers and that only seven of them are present in the latest Pharo class comment template. However, the information types proposed by the standard template tend to be present more often than other types of information. Additionally, we find that a substantial proportion of comments follow the writing style of the template in writing these information types, but they are written and formatted in a non-uniform way. This suggests the need to standardize the commenting guidelines for formatting the
Information types across Pharo projects
Class: FTTreeItem

I am an abstract class to define an Item use by a tree data source of Fast table.

Description

I define the basics methods needed by a FTTreeDataSource.
I use FTTreeItem to manage my elements and I am use by a FTFastTable.

Public API and Key Messages

• #data. anObject from: aFTTreeDataSource

This is my constructor that is use by FTTreeDataSource and myself

Example

Should not be instanciate.

Internal Representation and Key Implementation Points.

Instance Variables

dataSource: I am the dataSource that holds this Item.
children: I am a collection of Items calculate by the item. I contains the children of the Item.
Collected data

Class: FTTreeItem

I am an abstract class to define an Item used by a tree data source of Fast table.

Description
-------------------------------------------------
I define the basics methods needed by a FTTreeDataSource.
I use FTTreeItem to manage my elements and I am used by a FTFastTable.

Public API and Key Messages
-------------------------------------------------
• #data. anObject from: aFTTreeDataSource

This is my constructor that is used by FTTreeDataSource and myself.

Example
-------------------------------------------------

Should not be instanciate.

Internal Representation and Key Implementation Points.
-------------------------------------------------

Instance Variables
dataSource: I am the dataSource that holds this Item.
children: I am a collection of Items calculated by the item. I contains the children of the Item.
Identification of heuristics

Class: FTTreeItem

I am an abstract class to define an Item used by a tree data source of Fast table.

Description
-----------------------------------------------------------------
I define the basics methods needed by a FTTreeDataSource. I use FTTreeItem to manage my elements and I am use by a FTFastTable.

Public API and Key Messages
-----------------------------------------------------------------
• #data. anObject from: aFTTreeDataSource

This is my constructor that is use by FTTreeDataSource and myself

Example
-----------------------------------------------------------------
Should not be instanciate.

Internal Representation and Key Implementation Points.
-----------------------------------------------------------------

Instance Variables
dataSource: I am the dataSource that holds this Item. children: I am a collection of Items calculate by the item. I contains the children of the Item.
Taxonomy
Techniques
Training & Testing
Workflow

Taxonomy Definition

Automated Classification of Comments Types
Comment Evolution
Analyzing the co-evolution of comments and source code

Beat Fluri · Michael Würsch · Emanuel Giger · Harald C. Gall

Published online: 26 March 2009
© Springer Science+Business Media, LLC 2009

Abstract Source code comments are a valuable instrument to preserve design decisions and to communicate the intent of the code to programmers and maintainers. Nevertheless, commenting source code and keeping comments up-to-date is often neglected for reasons of time or programmers obliviousness. In this paper, we investigate the question whether developers comment their code and to what extent they add comments or adapt them when they evolve the code. We present an approach to associate comments with source code entities to track their co-evolution over multiple versions. A set of heuristics are used to decide whether a comment is associated with its preceding or its succeeding source code entity. We analyzed the co-evolution of code and comments in eight different open source and closed source software systems. We found with statistical significance that (1) the relative amount of comments and source code grows at about the same rate; (2) the type of a source code entity, such as a method declaration or an if-statement, has a significant influence on whether or not it gets commented; (3) in six out of the eight systems, code and comments co-evolve in 90% of the cases; and (4) surprisingly, API changes and comments do not co-evolve but they are re-documented in a later revision. As a result, our approach enables a quantitative assessment of the commenting process in a software system. We can, therefore, leverage the results to provide feedback during development to increase the awareness of when to add comments or when to adapt comments because of source code changes.
Java code comment co-evolution

Source: Analyzing the co-evolution of comments and source code, fig 1
Java code comment co-evolution

Analyzing the co-evolution of comments and source code

Beat Fluri · Michael Würsch · Emanuel Giger · Harald C. Gall

Published online: 26 March 2009
© Springer Science+Business Media, LLC 2009

Abstract  Source code comments are a valuable instrument to preserve design decisions and to communicate the intent of the code to programmers and maintainers. Nevertheless, commenting source code and keeping comments up-to-date is often neglected for reasons of time or programmers obliviousness. In this paper, we investigate the question whether developers comment their code and to what extent they add comments or adapt them when they evolve the code. We present an approach to associate comments with source code entities to track their co-evolution over multiple versions. A set of heuristics is used to decide whether a comment is associated with its preceding or its succeeding source code entity. We analyzed the co-evolution of code and comments in eight different open source and closed source software systems. We found with statistical significance that (1) the relative amount of comments and source code grows at about the same rate; (2) the type of a source code entity, such as a method declaration or an if-statement, has a significant influence on whether or not it gets commented; (3) in six out of the eight systems, code and comments co-evolve in 90% of the cases; and (4) surprisingly, API changes and comments do not co-evolve but they are re-documented in a later revision. As a result, our approach enables a quantitative assessment of the commenting process in a software system. We can, therefore, leverage the results to provide feedback during development to increase the awareness of when to add comments or when to adapt comments because of source code changes.

- Over 50% of the comment changes are related to source code changes
- Newly added code gets barely commented
- Growth factor of code and comments are equal over time
Pharo class comment co-evolution

What do class comments tell us? An investigation of comment evolution and practices in Pharo Smalltalk

Pooja Rani · Sebastiano Panichella · Manuel Leuenberger · Mohammad Ghafari · Oscar Nierstrasz

Received: date / Accepted: date

Abstract Previous studies have characterized code comments in various programming languages, and have shown how a high quality of code comments is crucial to support program comprehension activities and to improve the effectiveness of maintenance tasks. However, very few studies have focused on the analysis of the information embedded in code comments. None of them has compared developer practices to write comments following the standard guidelines or analyzed these characteristics in the Pharo Smalltalk environment.

These class commenting practices have their origins in Smalltalk-80, going back 40 years. Smalltalk traditionally separates class comments from source code, and offers a brief template for entering a comment for newly-created classes. These templates have evolved over the years, particularly in the Pharo environment. This paper reports the first empirical study investigating commenting practices in Pharo Smalltalk. As a first step, we analyze class comment evolution over seven Pharo versions. Then, we quantitatively and qualitatively analyze class comments of the most recent version of Pharo, to investigate the information types of Pharo comments. Finally, we study the adherence of developer commenting practices to the class template over Pharo versions.

The results of this study show that there is a rapid increase in class comments in the initial three Pharo versions, while in subsequent versions developers added comments to both new and old classes, thus maintaining a similar ratio. In addition, the analysis of the semantics of the comments from the latest Pharo version suggests that 23 information types are typically embedded in class comments by developers and that only seven of them are present in the latest Pharo class comment template. However, the information types proposed by the standard template tend to be present more often than other types of information. Additionally, we find that a substantial proportion of comments follow the writing style of the template in writing these information types, but they are written and formatted in a non-uniform way. This suggests the need to standardize the commenting guidelines for formatting the
Pharo Comment Evolution
Pharo class comment evolution

![Pharo class comment evolution chart]

- **New classes**
- **Old classes unchanged**
- **Old classes only comment changed**
- **Old classes only code changed**
- **Old classes comment and code changed**
Pharo class comment co-evolution

What do class comments tell us? An investigation of comment evolution and practices in Pharo Smalltalk

Pooja Rani · Sebastian Panichella · Manuel Leuenberger · Mohammad Ghafari · Oscar Nierstrasz

Received: date / Accepted: date

Abstract Previous studies have characterized code comments in various programming languages, and have shown how a high quality of code comments is crucial to support program comprehension activities and to improve the effectiveness of maintenance tasks. However, very few studies have focused on the analysis of the information embedded in code comments. None of them has compared developer practices to write comments following the standard guidelines or analyzed these characteristics in the Pharo Smalltalk environment.

These class commenting practices have their origins in Smalltalk-80, going back 40 years. Smalltalk traditionally separates class comments from source code, and offers a brief template for entering a comment for newly-created classes. These templates have evolved over the years, particularly in the Pharo environment. This paper reports the first empirical study investigating commenting practices in Pharo Smalltalk. As a first step, we analyze class comment evolution over seven Pharo versions. Then, we quantitatively and qualitatively analyze class comments of the most recent version of Pharo, to investigate the information types of Pharo comments. Finally, we study the adherence of developer commenting practices to the class template over Pharo versions.

The results of this study show that there is a rapid increase in class comments in the initial three Pharo versions, while in subsequent versions developers added comments to both new and old classes, thus maintaining a similar ratio. In addition, the analysis of the semantics of the comments from the latest Pharo version suggests that 23 information types are typically embedded in class comments by developers and that only seven of them are present in the latest Pharo class comment template. However, the information types proposed by the standard template tend to be present more often than other types of information. Additionally, we find that a substantial proportion of comments follow the writing style of the template in writing these information types, but they are written and formatted in a non-uniform way. This suggests the need to standardize the commenting guidelines for formatting the

- Over 50% of the comment changes are related to source code changes
- Newly added code gets commented often
- Growth factor of code and comments are not equal over time
Automatic generation and summarization of comments
Code summaries

- “Automatically generated, short, yet accurate descriptions of source code entities”.

- They give more information than just the header or the name of an artifact.

- Significantly shorter and faster to read than the source code they summarize.
Example of natural language summaries

Step 1
Type or paste your text into the box.

Albert Einstein was born on March 14, 1879 in Ulm, the first child of the Jewish couple Hermann and Pauline Einstein, née Koch. In June 1880 the family moved to Munich where Hermann Einstein and his brother Jakob founded the electrical engineering company Einstein & Cie. Albert Einstein’s sister Maria, called Maja, was born on November 18, 1881. Einstein’s childhood was a normal one, except that to his family’s irritation, he learnt to speak at a late age. Beginning in 1884 he received private education in order to get prepared for school. 1885 he started learning to play violin. Beginning in 1885 he received his primary education at a Catholic school in Munich (Peterschule); in 1888 he changed over to the Luitpold-Gymnasium, also in Munich. However, as this education was not to his liking and, in addition, he did not get along with his form-master he left this school in 1894 without a degree and joined his family in Italy where they had settled meanwhile.

In order to be admitted to study at the “Eidgenössische Polytechnische Schule” (later renamed ETH) in Zurich, Einstein took his entrance examination in October 1895. However, some of his results were insufficient and, following the advice of the rector, he attended the “Kantonsschule” in the town of Aarau in order to improve his knowledge. In early October 1896 he received his school-leaving certificate and shortly thereafter enrolled at the Eidgenössische Polytechnische Schule with the goal of becoming a teacher in Mathematics and Physics. Einstein, being an average student, finished his studies with a diploma degree in July 1900. He then applied, without success, for assistantships at the Polytechnic School and other universities. Meanwhile he had abandoned the German citizenship and formally applied for the Swiss one which he was granted on February 21, 1901.

Step 2
Drag the slider, or enter a number in the box, to set the percentage of text to keep in the summary.

35 %
Example of natural language summaries

**Step 1**
Type or paste your text into the box below.

Albert Einstein was born on March 14, 1879 in Ulm, the first child of the Jewish couple Hermann Einstein and Pauline Einstein, née Koch. In June 1880 the family moved to Munich where Hermann Einstein and his brother Jakob founded the electrical engineering company "Einstein & Cie." However, as this education was not to his liking and, in addition, he did not get along with his form-master he left this school in 1894 without a degree and joined his family in Italy where they had settled meanwhile.

In order to be admitted to study at the "Eidgenössische Polytechnische Schule" (later renamed ETH) in Zurich, Einstein took his entrance examination in October 1895. However, some of his results were insufficient and, following the advice of the rector, he attended the "Kantonsschule" in the town of Aarau in order to improve his knowledge. In early October 1896 he received his school-leaving certificate and shortly thereafter enrolled at the Eidgenössische Polytechnische Schule with the goal of becoming a teacher in Mathematics and Physics.

Einstein, being an average student, finished his studies with a diploma degree in July 1900. He then applied, without success, for assistantships at the Polytechnische Schule and other universities. Meanwhile he had abandoned the German citizenship and formally applied for the Swiss one which he was granted on February 21, 1901.

**Step 2**
Drag the slider, or enter a number in the box, to set the percentage of text to keep in the summary.

35%

**Step 3**
Read your summarized text. If you would like a different summary, repeat Step 2. When you are happy with the summary, copy and paste the text into a word processor, or text to speech program, or language translation tool.

Albert Einstein was born on March 14, 1879 in Ulm, the first child of the Jewish couple Hermann and Pauline Einstein, née Koch. In June 1880 the family moved to Munich where Hermann Einstein and his brother Jakob founded the electrical engineering company Einstein & Cie. However, as this education was not to his liking and, in addition, he did not get along with his form-master he left this school in 1894 without a degree and joined his family in Italy where they had settled meanwhile.

In order to be admitted to study at the "Eidgenössische Polytechnische Schule" (later renamed ETH) in Zurich, Einstein took his entrance examination in October 1895. However, some of his results were insufficient and, following the advice of the rector, he attended the "Kantonsschule" in the town of Aarau in order to improve his knowledge. In early October 1896 he received his school-leaving certificate and shortly thereafter enrolled at the Eidgenössische Polytechnische Schule with the goal of becoming a teacher in Mathematics and Physics.

Einstein, being an average student, finished his studies with a diploma degree in July 1900. He then applied, without success, for assistantships at the Polytechnische Schule and other universities. Meanwhile he had abandoned the German citizenship and formally applied for the Swiss one which he was granted on February 21, 1901.

https://www.textcompactor.com/
MILAN, Italy, April 18. A small airplane crashed into a government building in heart of Milan, setting the top floors on fire, Italian police reported. There were no immediate reports on casualties as rescue workers attempted to clear the area in the city's financial district. Few details were available about it immediately set off fears that it might be a terrorist act akin to the Sept. 11 attacks in the United States. Those fears sent U.S. stocks tumbling to session lows in late morning trading.

Witnesses reported hearing a loud explosion from a nearby office building, which houses the administrative offices of the local Lombardy region and sits next to the city's central train station. Italian state television said the crash put a hole in the 25th floor of the Pirelli building. News reports said smoke poured from the opening. Police and ambulances rushed to the building in downtown Milan. No further details were immediately available.
Comment summaries

Text Compactor
Free Online Automatic Text Summarization Tool

Follow these simple steps to create a summary of your text.

**Step 1**
Type or paste your text into the box.

Class: FTTreeItem
I am an abstract class to define an Item use by a tree data source of Fast table.

Description

---

I define the basics methods needed by a FTTreeDataSource.
I use FTTreeItem to manage my elements and I am use by a FTFastTable.

Public API and Key Messages

---

* `#data. anObject from: aFTTreeDataSource`
This is my constructor that is use by FTTreeDataSource and myself

Example

---

---

**Step 2**
Drag the slider, or enter a number in the box, to set the percentage of text to keep in the summary.

50%

**Step 3**
Read your summarized text. If you would like a different summary, repeat Step 2. When you are happy with the summary, copy and paste the text into a word processor, or [text to speech program](https://www.textcompactor.com/), or [language translation tool](https://www.textcompactor.com/)

Class: FTTreeItem
I am an abstract class to define an Item use by a tree data source of Fast table.

Description

---

I define the basics methods needed by a FTTreeDataSource.
I use FTTreeItem to manage my elements and I am use by a FTFastTable.

Public API and Key Messages

---

* `#data.`
Navigating classes

```java
package net.sourceforge.atunes.kernel.modules.repository.audio;

import java.io.File;

public final class AudioFile implements AudioObject, Serializable, Comparable<AudioFile> {
    private static final long serialVersionUID = -1139001443603556703L;
    private static transient Logger logger = new Logger();
    private File file;
    protected Tag tag;
    private List<File> externalPictures;
    protected long duration;
    protected long bitrate;
    protected int frequency;
    protected long readTime;
    private int stars = 0;

    public AudioFile(String fileName) {
        readFile(new File(fileName));
    }

    private void readFile(File file) {
        this.file = file;
        if (!isApeFile(file) && !isMPCFile(file)) {
            introspectTags();
            readAudioProperties(this);
        }
        this.readTime = System.currentTimeMillis();
    }
}
```

We look at:
- **Name of the class**
- **Attributes**
- **Methods**
- **Dependencies between classes**
Java class summaries

JSummarizer: An Automatic Generator of Natural Language Summaries for Java Classes

Laura Moreno¹, Andrian Marcus¹, Lori Pollock², K. Vijay-Shanker²

¹Department of Computer Science
Wayne State University
Detroit, MI, USA
{lmoreno, anurarcus}@wayne.edu

²Computer and Information Sciences Department
University of Delaware
Newark, DE, USA
{pollock, vijay}@cis.udel.edu

Abstract—JSummarizer is an Eclipse plug-in for automatically generating natural language summaries of Java classes. The summary is based on the stereotype of the class, which implicitly encodes the design intent of the class and is automatically inferred by JSummarizer. The tool uses a set of predefined heuristics to determine what information will be reflected in the summary, and it uses natural language processing and generation techniques to form the summary. The generated summaries can be used to re-document the code and to help developers to easier understand large and complex classes.

I. INTRODUCTION

During software evolution, depending on the task at hand, developers need to understand relevant parts of the code. In consequence, developers often spend more time reading code [1] than writing it. Good leading comments help when reading code, by providing developers with at least a superficial understanding of the source code artifact that they describe. However, outdated or missing comments are very common and developers often must read more of the code or turn to external documentation in order to gain any understanding of the code relevant to their task.

An obvious solution to this problem would be enforcing the creation and continuous update of internal documentation. While such a solution may work with new code, it will likely not work on existing, poorly-documented code. A more suitable approach is automatically generating summaries that describe the code. Such summaries can be used for re-

- Generate class summaries
- Decide the information to generate
- Gather the heuristics
- Generate the info
Workflow

JSummarizer: An automatic generator of natural language summaries for Java classes by Moreno
Summary

A TreeModel<EventListener, ItemListener> implementation, and DefaultTreeModel extension for explorer tree models. This class controls external objects, i.e., the majority of its methods are controllers or object creators.

It allows managing:
- children; and
- structure changed.

It also allows:
- handling model element changed;
- removing node from parent;
- handling model element added;
- handling model element removed; and
- inserting node into new child mutable tree node.

This class declares the helper class ExplorerUpdater.
Questions when summarizing classes

- What information to include in the summaries?
- How much information to include in the summaries?
- How to generate and present the summaries?
Content adequacy vs. expressiveness

- Missing some information
- Not easy to understand
To overcome these limitations…

Researchers proposed to detect source code descriptions from external sources:

- Mailing list and issue trackers
- StackOverflow discussions
Linguistic Analysis
Linguistic analysis for English

TAACO
Linguistic analysis for English
Linguistic analysis for English
Style analysis of Java comments

Automatic Quality Assessment of Source Code Comments: The JavadocMiner

Ninus Khamis, René Witte, and Juergen Rilling
Department of Computer Science and Software Engineering
Concordia University, Montréal, Canada

Abstract. An important software engineering artefact used by developers and maintainers to assist in software comprehension and maintenance is source code documentation. It provides insights that help software engineers to effectively perform their tasks, and therefore ensuring the quality of the documentation is extremely important. Inline documentation is at the forefront of explaining a programmer's original intentions for a given implementation. Since this documentation is written in natural language, ensuring its quality needs to be performed manually. In this paper, we present an effective and automated approach for assessing the quality of inline documentation using a set of heuristics, targeting both quality of language and consistency between source code and its comments. We apply our tool to the different modules of two open source applications (ArgoUML and Eclipse), and correlate the results returned by the analysis with bug defects reported for the individual modules in order to determine connections between documentation and code quality.

1 Introduction

“Comments as well as the structure of the source code aid in program understanding and therefore reduce maintenance costs.” – Elshoff and Marcotty (1982) [1]
Style analysis of Java comments

Quality Analysis of Source Code Comments

Daniela Steidl  Benjamin Hummel  Elmar Juergens
CQSE GmbH, Garching b. München, Germany
{steidl,hummel,juergens}@cqse.eu

Abstract—A significant amount of source code in software systems consists of comments, i.e., parts of the code which are ignored by the compiler. Comments in code represent a main source for system documentation and are hence key for source code understanding with respect to development and maintenance. Although many software developers consider comments to be crucial for program understanding, existing approaches for software quality analysis ignore system commenting or make only quantitative claims. Hence, current quality analyses do not take a significant part of the software into account. In this work, we present a first detailed approach for quality analysis and assessment of code comments. The approach provides a model for comment quality which is based on different comment categories.

To categorize comments, we use machine learning on Java and C/C++ programs. The model comprises different quality aspects: by providing metrics tailored to suit specific categories, we show how quality aspects of the model can be assessed. The validity of the metrics is evaluated with a survey among 16 experienced software developers, a case study demonstrates the relevance of the metrics in practice.

I. INTRODUCTION

A significant amount of source code in software systems consists of comments, which document the implementation and help developers to understand the code, e.g., for later modification or reuse. Several researchers have conducted experiments showing that commented code is easier to understand than code without comments [1], [2]. Comments are the second most-used documentary artifact for code understanding, behind only the code itself [3]. In addition, source code documentation is also vital in maintenance and forms an important part of the general documentation of a system. In contrast to external documentation, comments in source code as they do not enhance system understanding and quantitative measures cannot detect outdated/ useless comments.

Furthermore, a complete model of comment quality does not exist. Coding conventions, e.g., marginally touch on the topic of commenting code but mostly lack depth and precision [8]. So far, (semi-) automatic methods for comment quality assessment have not been developed as comment analysis is a difficult task: Comments comprise natural language and have no mandatory format aside from syntactic delimiters. Hence, algorithmic solutions will be heuristic in nature.

Problem Statement. Current quality analysis approaches ignore system commenting or are restricted to the comment ratio metric only. Hence, a major part of source code documentation is ignored during software quality assessment.

Contribution. Based on comment classification, we provide a semi-automatic approach for quantitative and qualitative evaluation of comment quality.

We present a semi-automatic approach for comment quality analysis and assessment. First, we perform comment categorization both for Java and C/C++ programs based on machine learning to differentiate between different comment types. Comment categorization enables a detailed quantitative analysis of a system’s comment ratio and a qualitative analysis tailored to suit each single category. Comment categorization is the underlying basis of our comprehensive quality model. The model comprises quality attributes for each comment category based on four criteria: consistency throughout the project, completeness of system documentation, coherence with source
Future work

- Evaluate if a comment is good or not
- Detect inconsistent comments
- Propose refactoring of comments
Summary

Understanding code...

Code comment types
- Documentation (/** … */)
  (also used for packages / classes / methods)
- Block comments (/* … */)
- Inline comments (//…)

What is a good comment?
- Helps other developers in working with your code
- Describes why, and not how
- Reveals intent, limitation, assumptions, design decisions
- Justifies the violation of a programming style

Comment analysis tools
- Syntax
- Semantics
- Style

Workflow