Use-Case-Specific Source-Code Documentation for Feature-Oriented Programming

Sebastian Krieter
University of Magdeburg, Germany
sebastian.krieter@st.ovgu.de

Reimar Schröter
University of Magdeburg, Germany
reimar.schroeter@ovgu.de

Wolfram Fenske
University of Magdeburg, Germany
wolfram.fenske@ovgu.de

Gunter Saake
University of Magdeburg, Germany
gunter.saake@ovgu.de

ABSTRACT

Source-code documentation is essential to efficiently develop and maintain large software products. Documentation is equally important for *software product lines (SPLs)*, which represent a set of different products with a common code base. Unfortunately, proper support for documenting the source code of an SPL is currently lacking, because source code variability is not considered by current documentation tools. We introduce a method to provide source-code documentation for feature-oriented programming and aim to support developers who implement, maintain, and use SPLs. We identify multiple use cases for developers working with SPLs and propose four different documentation types (meta, product, feature, and context) that fulfill the information requirements of these use cases. Furthermore, we design an algorithm that enables developers to create tailor-made documentation for each use case. Our method is based on the documentation tool Javadoc and allows developers to easily write documentation comments that contain little overhead or redundancy. To demonstrate the efficiency of our method, we present a prototypical implementation and evaluate our method with regard to documentation effort for the SPL developers by documenting two small SPLs.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*documentation*

General Terms

Documentation

Keywords

Feature-Oriented Programming, Software Product Lines, Source Code Documentation, API Documentation

VaMoS '15 January 21 - 23 2015, Hildesheim, Germany
Copyright 2015 ACM 978-1-4503-3273-6/15/01 ...$15.00
http://dx.doi.org/10.1145/2701319.2701329

1. INTRODUCTION

To reduce the amount of implementation effort for a software product, developers try to reuse as much source code as possible from earlier and similar products [10]. *Software product line engineering (SPLE)* is a concept aimed at achieving this reuse in a systematic way. The idea is that one SPL contains many different variants of one product, which all share a common code base [6, 11]. One promising technique to implement SPLs is *feature-oriented programming (FOP)*, which divides an SPL into single features and stores the source code for each feature locally separated [12]. By combining a subset of all given features, different products can be generated.

On the one hand, the modularization provided by FOP makes it easier to identify the code sections that implement a given feature. On the other hand, however, it becomes more challenging for a developer to get an overview of all the code that influences the behavior of a specific class or method. Consider the simple example for the method `receive` of a chat SPL for two different features, given in Figure 1. When calling this method, the developer has to reason when it is necessary to check the result for `null` values. With good documentation, this can be avoided as the relevant information is directly linked to the method (e.g., as tooltip). Thus, good documentation of the source code is needed to prevent bugs and loss of development time.

In general, source-code documentation is an important part of software development, because it is necessary for the understanding of source code during and after the development process. But especially when trying to use or reuse source code that was written by other developers, documentation of such code becomes a crucial aspect for program comprehension. The most common way of documenting source code of high level programming languages is the use

```
private Message receive(Message msg) {
  if (msg.isEncoded())
    msg.decode();
  return original(msg);
}                                    Encryption
```

```
private Message receive(Message msg) {
  if (SpamFilter.filter(msg))
    return null;
  return original(msg);
}                                    Spam
```

Figure 1: Method `receive` in two different features

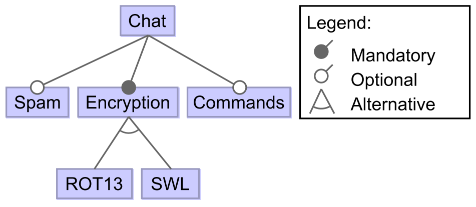

Figure 2: Feature model excerpt of the Chat SPL

of documentation tools like Javadoc[1] or Doxygen[2].

Currently, Javadoc and other documentation tools are unaware of the concept of FOP, which prevents them to unfold their full potential of helping developers of SPLs. Hence, we propose an extension for Javadoc, which can also be adapted for other documentation tools. We aim to support developers implementing and maintaining SPLs, as well as developers using SPL products or reusing (parts of) the source code of an SPL. Experience indicates that developers need different information for each of these use cases. Thus, we design different documentation types to match the specific information needs accordingly. Our proposed solution consists of four documentation types that enable the dynamic generation of source-code documentation for every *product*, every *feature*, a whole *software product line*, and all possible *context interfaces*, which are a new kind of development support for SPLs [14] (see Section 3). For this purpose, developers of a feature-oriented SPL are able to write documentation comments for their code consistently with the FOP design by dividing information into feature-independent and feature-specific parts (see Section 4) and generate the documentation for each type on demand.

To evaluate our proposed method, we have implemented it for FeatureHouse[3] [2] product lines and Javadoc. However, due to its modular design, our concept can be adapted to other FOP implementations (e.g., AHEAD[4] [4]) and other common documentation languages (e.g., Doxygen).

In summary, we make the following contributions:

- We identify and describe four documentation types for SPLs with individual information requirements.
- We present a method for generating tailor-made source-code documentation for all uses cases.
- We evaluate our method in terms of documentation effort by comparing it to straightforward approaches for single documentation types.

2. BACKGROUND

To clarify the challenges of documenting an SPL, this section provides background information on FOP and the approach to *application programming interface (API)* documentation taken by Javadoc.

2.1 Feature-Oriented Programming

The basic principle of FOP is the decomposition of all components of an SPL into a set of features [1]. We consider a feature in conformity with Kang et al. as "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems" [8]. Features of an SPL can be described with a feature model that defines dependencies between the features, thereby specifying which *configurations* of the SPL are valid. In Figure 2, we display an excerpt of the feature model of the Chat SPL, which implements multiple products of a simple chat application. The model specifies the two core features *Chat* (root feature) and *Encryption* (mandatory child of *Chat*), which are part of every valid configuration. Additionally, the model specifies the four non-core features *ROT13* and *SWL* (alternative features) and *Commands* and *Spam* (optional features).

To generate a certain product, all features selected in a configuration are combined by using the process of superimposition, which works on *feature structure trees (FSTs)* [3]. In general, all components of a feature can be combined by using FSTs and superimposition. This is called the *principle of uniformity* [5].

We provide an example of FSTs and superimposition for Java source code in Figure 3. The package name `client` is used as the first level of the tree. All nodes on the second level are classes (`Client`), and their children are class members (`send`, `receive`, and `encrypt`). When generating a product, all FSTs for the given feature subset (configuration) are combined by superimposition in a certain order, defined by the developer. For this, superimposition recursively composes all the nodes of two FSTs on the same level with an equal name and type [3]. In our example, such nodes are `client` (package), `Client` (class), and `send` (method). Nodes that only exist in one of the two FSTs (`receive` and `encrypt`) are added to the resulting FST.

Inside of feature source code, the developer can define new classes and class members or refine classes and class members of other features. In Figure 1, we give an example of method refinement in FOP. The method `receive` is originally defined in *Chat* and is later refined in *Encryption* (Lines 1-5) and *Spam* (Lines 6-10) by using the keyword `original`. If a generated product contains both features, the implementation of `receive` is taken from *Spam*, and `original` in *Spam* is replaced with the implementation from feature *Encryption*. Additionally, the `original` given in *Encryption* is replaced with the original implementation from *Chat*.

2.2 Javadoc

In this paper, we consider source-code documentation for developers that implement, maintain, or use SPLs. An established way to create source-code documentation is the use of tools such as *Javadoc*, which generates API documentation from documentation comments in the source code. Every source-code element (package, class, method, or field) that should be documented is provided with a documenta-

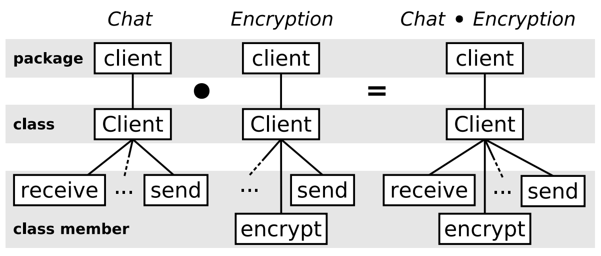

Figure 3: Example of the superimposition of two FSTs for the Chat SPL (adopted from [3])

[1]http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html
[2]http://www.stack.nl/~dimitri/doxygen/
[3]http://www.infosun.fim.uni-passau.de/spl/apel/fh/
[4]https://www.cs.utexas.edu/users/schwartz/ATS.html

```
1  /** Provides methods for encrypting
2   * and decrypting strings.
3   * @author Author 1
4   * @author Author 2
5   * @version 1.0
6   */
7  public class Encryption {
8    /** Encrypts a string.</br>
9     * Uses the ROT13 algorithm.
10    * @param text the plaintext
11    * @return the encrypted string
12    * @see Encryption#decrypt(String)
13    */
14   public static String encrypt(String text) {...}
15   /** ... */
16   public static String decrypt(String code) {...}
17 }
```

Figure 4: Example of Javadoc comments

tion comment, which holds all relevant information for documenting this element. In Javadoc, a documentation comment consists of a description and a number of block tags. In the remainder of the paper, we use the term *comment* to refer to a documentation comment. The description of a comment provides loose information about the purpose, functionality, and implementation details for a source-code element. Each block tag of a comment gives information for a certain aspect of the source-code element (e.g., author, return value). Therefore, some block tags can occur more than once in a comment (e.g., multiple authors).

An example of two comments can be seen in Figure 4. The source code provides a comment for the class `Encryption` (Lines 1-6) and for the method `encrypt` (Lines 8-13). The *description* of both comments (Lines 1-2 and 8-9) states the general purpose of the source-code elements. The block tags (Lines 3-5 and 10-12), in turn, provide information about the *source-code author* (Lines 3-4), *current version* (Line 5), *parameter* (Line 10), and *return value* (Line 11) of method `encrypt`. A comment can also contain *references* to comments of other source-code elements (Line 12).

3. REQUIRED DOCUMENTATION

We have identified several use cases for developers working with SPLs, which greatly benefit from tailor-made source-code documentation. The main use cases are the implementation, debugging, and maintenance of an SPL's functionality. Furthermore, in some cases, it is useful to reuse single features in other SPLs or use whole SPLs (e.g., in multi product lines [7]). Another important use case is the usage of generated products in other software (e.g., as libraries).

The existence of multiple use cases points out that the documentation output provided by a documentation generator has to be tailored to fit the information requirements of different use cases. We identified four different documentation types that cover the information needs of all use cases mentioned above. These types are the documentation of *SPLs*, *products*, *features*, and *context interfaces*.

We exemplify the documentation types with the help of the method `send` from the Chat SPL. In Figure 5, we show `send` in the features *Chat* and *Commands*. Both features contain a full Javadoc comment that provides all available information for the respective feature. Since it is not important for our approach whether a comment addresses a class, method, or field, we use the term *signature* to refer to classes, methods, and fields.

```
/** Sends a message to the Server.</br>
 * Creates a new {@link TextMessage} and
 * sends it to the server.</br>
 * @param line the message content.</br>
 *   The message may contain any character.
 */
public static void send(String line) {
  if (canSend())
    sendObject(toTextMessage(line));
}                                          Chat
```

```
/** Sends a message to the Server.</br>
 * Can be used to trigger user commands.
 * @param line the message content.</br>
 *   If the message starts with a /, the
 *   whole line is interpreted as user command.
 */
public static void send(String line) {
  if (line.startsWith("/"))
    // handle command ...
  else
    original(line);
}                                      Commands
```

Figure 5: Method send in features *Chat* and *Commands* with complete Javadoc comments

```
/** Sends a message to the Server.
 * @param line the message content.
 */
public static void send(String line) {...}
```

Figure 6: Meta documentation

Meta Documentation.

When maintaining or reusing an SPL, it is useful to get a first overview of the functionality and variability. Especially developers that are new to an SPL project need this information to do their work properly. For this use case, the *meta documentation* is most suitable. In this documentation type, every signature of an SPL is explained in general terms. Specifically, the meta documentation describes the general purpose and any other *feature-independent* information for all signatures, regardless of the feature in which a signature is defined. Details about implementations that are specific to a certain feature (*feature-specific* information) are not contained in this documentation type.

In Figure 6, we present a comment for the meta documentation of the Chat SPL. Note that it exclusively contains feature-independent information for the method `send`.

The challenge of this documentation type is that it contains exactly one comment for every signature. Due to the concept of FOP, signatures can be defined multiple times in different features, which leads to the problem of merging different comments for one signature. The straightforward approach to this problem is to provide a separate documentation for the whole SPL (i.e., not a direct comment for each signature). This approach is feasible because there is exactly one meta documentation for an SPL. A disadvantage of this approach is the separation of source code and documentation, which can easily lead to inconsistencies. With regard to our other documentation types, it is also a problem that this separate documentation does not contain feature-specific information. Therefore, a separate meta documentation will be difficult to reuse for other documentation types.

Product Documentation.

Generated products of an SPL can be used in other software products (e.g., as libraries). To enable a developer to

```
/** Sends a message to the Server.</br>
 *  Creates a new {@link TextMessage} and
 *  sends it to the server.</br>
 *  Can be used to trigger user commands.
 *  @param line the message content.</br>
 *     If the message starts with a /, the
 *     whole line is interpreted as user command.
 */
public static void send(String line) {...}
```

Figure 7: Product documentation with included optional feature *Commands*

use such a generated product correctly, a documentation of its API is needed. This use case is covered with the *product documentation*. It provides all relevant information for a generated product and consists of feature-independent information as well as information specific to the product's configuration.

In Figure 7, we present comments for a product of the Chat SPL, which, in addition to the root feature *Chat*, contains the feature *Commands*. Consequently, the product documentation contains implementation details for both features (Lines 2-4 and 6-7).

Since there is no refinement mechanism for comments, like there is for the source code itself, the problem for generating the product documentation consists of merging comments for refined signatures (similar to the meta documentation). Thus, the straightforward approach is to introduce such a mechanism and use a keyword like `@original` inside of the comments. With this approach, all comments could be handled in accordance with the principle of uniformity. Again, this straightforward approach only considers one documentation type and is not applicable to others.

Feature Documentation.

Due to their modular structure, it is possible to reuse features in other SPL projects. This is not necessarily done by the same developer who implemented the feature in the first place. Thus, this use case needs a documentation type that provides information about the function volume of features and the SPL and also details about possible pitfalls in the concrete implementation. This can be achieved with the *feature documentation*, which documents all signatures defined (and refined) in one feature. Feature documentation provides both feature-independent and feature-specific information.

In Figure 5, we present example comments for the feature documentation of *Chat* and *Commands*. The comments contain feature-independent information (Lines 1, 4, 11, and 13) and details for the concrete implementation in the corresponding features (Lines 2-3, 5, 12, and 14-15). Furthermore, this example illustrates the merge problem for product documentation, as both comments contain contradictory information (Lines 5 and 14-15).

The straightforward approach to generate this documentation type is to provide separate documentation for every feature. Unfortunately, this solution produces lots of redundant data because feature-independent information is stored multiple times. Because of the high amount of redundancy, this approach is not practical for the generation of other documentation types.

Context-Interface Documentation.

A promising approach to ease maintenance and implementation tasks of SPLs based on FOP are *context interfaces*, which help developers identify all signatures that are safely accessible from a given source code position [14]. Documentation tailored to these interfaces would allow developers to profit even more from this functionality. This documentation type is of special interest to developers currently implementing an SPL. Context interfaces give a special view on an SPLs source code. All signatures that are safely accessible within a certain feature context are contained in such an interface. Every signature contained in a context interface is included in this documentation with feature-independent as well as feature-specific information. Since the context documentation is used at development time, the concrete implementation of a signature is unknown. As an example, consider the method **`encrypt`** from Figure 4. The implementation of this method in a certain product depends on whether feature *ROT13* or *SWL* (see Figure 2) is part of the configuration. Therefore, the context-interface documentation provides information for all possible implementations of a signature, labeled with the corresponding feature.

```
/** Sends a message to the Server.</br>
 *  [Chat] Creates a new {@link TextMessage} and
 *    sends it to the server.</br>
 *  [Commands] Can be used to trigger
 *    user commands.</br>
 *  @param line the message content.</br>
 *    [Commands] If the message starts with
 *    a /, the whole line is interpreted as
 *    user command.
 */
public static void send(String line) {...}
```

Figure 8: Context-interface documentation

In Figure 8, we present an example comment for a context-interface documentation within the context of feature *Commands*. It contains feature-independent information (Lines 1 and 6) and implementation details for all features in which the method is defined (Lines 2-3, 4-5, and 7-9). Similar to the feature documentation, the straightforward approach for this documentation type is to provide context documentation for every feature. Again, this approach introduces a large amount of duplication of feature-independent as well as feature-specific information.

With our four introduced documentation types, we are able to fit the information requirements of all identified use cases. Furthermore, we presented four straightforward approaches that allow us to create documentation for each documentation type. However, we already stated the problem that the straightforward approaches are incompatible with each other because they lack meta information (i.e., information type, priority) about their comments. Thus, in the likely case that developers need documentation for more than one use case, the straightforward approaches require developers to create and maintain the input for each documentation type separately. This is not desirable as it leads to massive overhead and redundant information in the comments. Consequently, our goal is to have redundancy-free comments from which tailor-made documentation for all use cases can be generated. In order to achieve this goal, we need a new way to structure comments in SPL source code.

4. TAILORED API DOCUMENTATION

We extended the standard Javadoc structure to provide

```
 1  /**{@general 0}                                    Chat
 2   * Sends a message to the Server.
 3   * @param line the message content.
 4   */
 5  /**{@feature 0}
 6   * Creates a new {@link TextMessage} and
 7   * sends it to the server.
 8   * @param line The message may contain
 9   * any character.
10   */
11  public static void send(String line) {...}
```

```
12  /**{@feature 0}                                Commands
13   * Can be used to trigger user commands.
14   */
15  /**{@feature 1}
16   * @param line If the message starts with a /,
17   * the whole line is interpreted as user command.
18   */
19  public static void send(String line) {...}
```

Figure 9: Extended comment syntax in feature *Chat* and *Commands*

all required information. Based on this extended Javadoc structure, we use a generation algorithm to create tailor-made documentation for all use cases on demand.

4.1 Information Structuring

In order to tailor documentation to fit every documentation type, we divide comments into two distinct information types. In addition, for conflict resolution, we introduce priorities for every part of a documentation.

The main feature of our extended syntax is the distinction of information types, where *feature-specific information* is related to a certain feature and *feature-independent information* (i.e., the general purpose of a class, method or field) is independent from all feature implementations. Both types of information are explicitly separated in all comments, which helps reduce redundancy. Moreover, this separation enables us to handle both types of information differently, depending on the target documentation type (e.g., meta documentation only needs feature-independent information).

As stated in Section 3, comments can occur multiple times for a signature and thus, have to be merged together. Priorities are a suitable mechanism to control the merge process for comments. In case there are multiple comments with contradictory information, the SPL developer can decide which comment contains the right information.

4.2 Extended Javadoc

We introduce two new keywords, *general* and *feature*, to maintain the new comment structure. In order to assign an information type and priority to a comment, the developer has to use one of our new keywords at the very beginning of a comment. If a developer wants to specify different information types or priorities for a signature, they may use multiple comments.

We exemplify the usage of our keywords with the help of the method `send` in feature *Chat* and *Commands* (see Figure 9). The first comment in the example (Lines 1-4) provides a description and an `@param` tag (Lines 2-3). By using the keyword `{@general 0}` (Line 1), the description and the `@param` block tag inside this comment are feature-independent and have the lowest priority (0), which means that they will be overridden by any other feature-independent description or `@param` block tag with a higher priority. Feature-specific information with priority 0 is provided by the second comment (Lines 5-10), which uses the keyword `{@feature 0}`. The second comment holds additional information for the description and the `@param` tag. The source code in feature *Commands* contains two more comments for `send`. Since it is unnecessary to store feature-independent information redundantly, both comments in feature *Commands* contain only feature-specific information (Lines 12 and 15). The first comment in *Commands* (Lines 12-14) provides further information for the description with priority 0, whereas the second comment (Lines 15-18) provides further information for the `@param` tag with the higher priority 1. The higher priority is assigned due to the fact that this information contradicts the information given in *Chat* (Lines 8-9). Thus, the developer decided that the `@param` information in *Commands* is more important than the feature-specific `@param` information in *Chat* and will override it in configurations where both features are selected.

In case no keyword is given at a comment's beginning, the information in the comment will be treated as feature-independent with priority 0. Therefore, the generation algorithm is also able to work with the standard Javadoc structure.

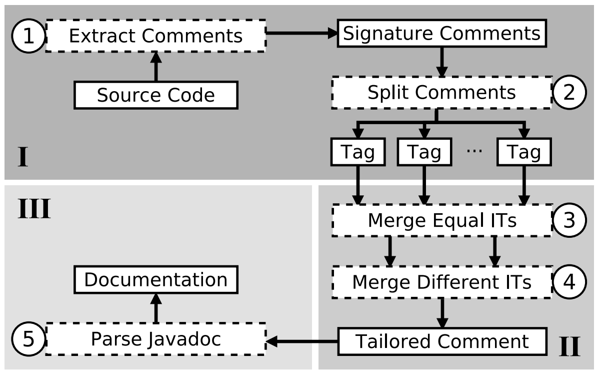

Figure 10: Basic process (main phases) of the generation algorithm (IT = Information type)

4.3 Generation Algorithm

Based on our extended Javadoc syntax for FOP, we propose an algorithm to generate all of our defined documentation types (see Section 3). As input, the developer selects a documentation type. Then, the algorithm uses a list of all relevant features and all relevant signatures in the source code to generate the documentation (e.g., as HTML) for the selected documentation type.

The generation algorithm has a modular design and consists of three phases. We depict the basic process of the three phases in Figure 10. The first phase (**I**) consists of two steps and reads all needed information from the source code. At first (Step ①), the algorithm extracts all relevant comments from the source code (i.e., comments from signatures and features contained in the documentation). Then, the algorithm splits the comments for each signature into a list of block tags (Step ②). The second phase (**II**) is done for each signature separately. To handle feature-independent and feature-specific information separately, the algorithm merges block tags in two consecutive steps (③, ④) and creates a tailored comment according to the desired documentation type. In the last phase (**III**), the Javadoc tool parses the tailored comments to create the final documen-

#	Source	Tag	IT	Prio
1	Line 2	Description	I	0
2	Line 3	`@param line`	I	0
3	Lines 6-7	Description	S	0
4	Lines 8-9	`@param line`	S	0
5	Line 13	Description	S	0
6	Lines 16-17	`@param line`	S	1
A	Line 2	Description	I	0
B	Line 3	`@param line`	I	0
C	Concatenate **#3**, **#5**	Description	S	0
D	Override **#4** with **#6**	`@param line`	S	1
I	Concatenate **#A**, **#C**	Description	-	-
II	Concatenate **#B**, **#D**	`@param line`	-	-

Table 1: Tag list for the send example, with raw input (first part), results after first merge (second part), and results after second merge (last part). (I = feature-independent, S = feature-specific)

tation (Step ⑤). The last phase has the advantage that the Javadoc tool is independent from the merge process, which allows the developer to use all the standard options of the Javadoc tool (e.g., choose an output format).

In the following, we explain the splitting of the block tags (②) and the two merge steps (③, ④) in further detail. For this, we use our example for the method `send` (see Figure 9) and demonstrate how our algorithm creates a product documentation for a product that contains the features *Chat* and *Commands*.

Split Comments.

All comments for a signature are read from the source code in the feature order specified by the developer (①). Afterwards, these comments are split into single block tags (②). The result of this splitting step is a list of all block tags assigned to a certain signature. This list also includes the comment description, which is treated as a single block tag. As an example, we list all tags for `send` in the first part of Table 1 (i.e., above the first double line). Note that the tag list is ordered by the feature in which the tags are specified because comments are read in the defined feature order (i.e., the same order that is used for superimposition in FOP). Since every comment follows the extended syntax, all tags have additional details about their information type, priority, and the feature in which they are defined.

To merge block tags, it is necessary that they can be identified uniquely within the comments of one signature. Hence, the algorithm internally assigns a unique identifier to each block tag. If block tags, by Javadoc convention, are not allowed to occur more than once, it is enough to use the tag name as identifier (e.g., `@return`). However, some block tags are allowed to occur multiple times in a comment and thus, require a more complex identifier. The identifier is constructed directly from the comment and consists of the tag name followed by a part of the tag's description (e.g., the parameter name for `@param` tags). In Table 2, we list which part of a block tag's description is used for the identifier, in addition to its name. Together with the information provided by our extended syntax, every block tag assigned to a certain signature is now uniquely identifiable.

Tag	Identifier Part	Merge Strategy
Description	-	concatenate
`@author`	author name	override
`@deprecated`	-	concatenate
`@param`	parameter name	concatenate
`@return`	-	concatenate
`@see`	signature name	override
`@serial`	field name	concatenate
`@serialData`	-	concatenate
`@serialField`	field name	concatenate
`@since`	-	override
`@throws`	exception name	concatenate
`@version`	-	override
Custom Tag	-	override

Table 2: Overview over identifier parts and merge strategies for Javadoc block tags

Merge Equal Information Types.

The first merge step (③) combines block tags that have the same identifier and the same information type. Tags with the same identifier and information type can occur multiple times for one signature, since they can differ in their priority and the feature in which they are defined.

There are two possible merge strategies for handling block tags that have to be merged. The tags can be *concatenated* or one tag can *override* the other. In order to decide how to merge the tags, first their priorities are compared. If the tags have different priorities, the tag with the higher priority overrides the content of the other tag. In case that both tags have the same priority, the action depends on the tag type. For some tags like `@version` it does not make sense to concatenate the contents. Thus, those tags get overridden. For other tags, such as `@param`, it is reasonable to keep the content of both tags and thus, those are concatenated. The default merge strategy is always applied in the specified feature order (i.e., preceding tags appear on top of concatenated comments or get overridden by following tags). In Table 2, we list the default merge strategy for all block tags.

We demonstrate the first merge step with the tag list of our example, listed in the first part of Table 1. Since they have equal identifiers and information types, the algorithm has to merge the tags #3 and #5 and the tags #4 and #6. Tags #3 and #5 have the same priority and thus, the algorithm takes the default merge strategy for comment descriptions and concatenates the contents of the tags. By contrast, the tags #4 and #6 differ in their priority and thus, #6 overrides the contents of #4. The result of the first merge step can be seen in the second part of Table 1 (i.e., between the two double lines).

Merge Different Information Types.

The second merge step (④) combines all block tags with equal identifiers and different information types. After the first merge step (③), all tags in the list with an equal identifier can only differ in their information type, which means that feature-independent and feature-specific information are now merged together. In case of the meta documentation, the algorithm only takes feature-independent and discards all feature-specific information. For all other documentation types, feature-independent and feature-specific information is concatenated in a way that all feature-independent information appears on top of a tag. Afterwards, the tag list is converted into a single comment by sorting the list according to Javadoc conventions and then concatenating all tags in the list. The second merge step also creates pseudo source code for the current signature, which is necessary for the Javadoc tool to work correctly. Pseudo source code contains all signature declarations needed for the documentation. However, it only contains empty method bodies and no field initializers. Finally, the merged comment is attached to the respective signature in the pseudo source code and the algorithm process the next signature.

In our example, for the tags in Table 1, the algorithm combines #A and #C to the description and #B and #D to the `@param` tag of the final comment, which can be seen in the last part of Table 1 (i.e., below the second double line). We already depicted the resulting tailored comment in Figure 7.

5. EVALUATION

Documenting an SPL with straightforward approaches requires much time and effort from the developers. Thus, our method aims to reduce this effort to a minimum. In our evaluation, we therefore compare the effort of creating and maintaining comments with our extended syntax to the straightforward approaches, which we presented in Section 3.

To evaluate our method, we made a prototypical implementation of our algorithm. The algorithm must be able to extract the signatures and their comments from the source code. Thus, we decided to implement it as an extension of FeatureIDE[5] [15], which is a plug-in for the popular *integrated development environment (IDE)* Eclipse. FeatureIDE covers the functionality of creating, managing, and analyzing SPLs and supports a variety of different implementation techniques for SPLs. With FeatureIDE and FeatureHouse SPLs, we are capable of providing the necessary input for our generation algorithm.

5.1 Evaluation Procedure

In our evaluation, we demonstrate the efficiency of our extended syntax in terms of overhead and redundancy. For all documentation types, we measure the amount of *input* that is necessary to produce the documentation with the straightforward approach and our generation algorithm. In detail, we compare the character count of the comments used for creating the documentation.

In Section 3, we stated that the straightforward approach for product documentation requires the design of a refinement mechanism for comments. Since this was not part of our research, we could only estimate the input size for this straightforward approach. The input comments for the product documentation must contain all available information, since this documentation type has to provide detailed documentation for each product. However, the respective straightforward approach is able to work with almost redundancy free comments. Therefore, we assumed that the input size of the straightforward approach for product documentation is approximately equal to our method. For all other straightforward approaches, we are able to generate the input comments with our generation algorithm. This is due to the fact that the created tailored comments are identical to the input comments for the straightforward approaches

[5]`http://fosd.de/fide`

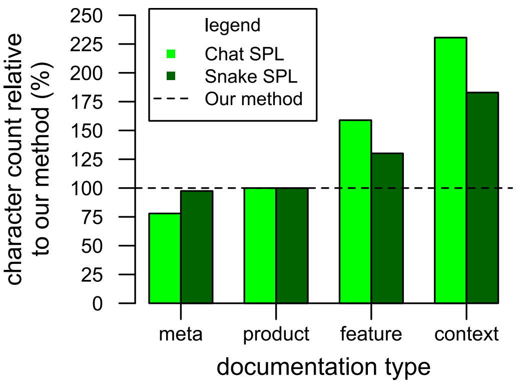

Figure 11: Barplot of the evaluation results, relative to our extended syntax

for meta, feature, and context documentation.

We documented two SPLs with our extended syntax in order to evaluate our approach. The first SPL is Chat, which we have already used in our examples. For the evaluation, we used the full Chat SPL, which contains 13 features and allows 120 valid configurations. The Chat SPL consists of 12 classes, 70 methods, and 47 fields. The second SPL is Snake with 21 features, which represents a total of 5580 variants of the classical video game Snake. The Snake SPL consists of 28 classes, 197 methods, and 133 fields. In contrast to Chat, we created Snake by decomposing a single product into features of an SPL. This includes the already existing comments, which we converted to our new syntax.

5.2 Evaluation Results

In Figure 11, we illustrate the results for both evaluated SPLs. The evaluation gives similar results for both SPLs. In both cases, the straightforward approach for the meta documentation needs slightly less input than our method. By contrast, if we use our method for feature and context documentation, we need much less input than the straightforward approaches.

Meta documentation does not contain feature-specific information, which explains the low result for the straightforward approach. The high amount of required input for feature and context documentation is mostly explained by the feature-independent information. As stated in Section 3, both documentation types provide a complete documentation for every feature. This leads to a high amount of redundant feature-independent and feature-specific information.

The evaluation results underline the benefit of our approach to separate feature-independent from feature-specific information. Furthermore, in most cases, developers want to use all documentation types for an SPL. If developers were to use all straightforward approaches instead of our method, this would lead to an approximately fourfold increase in documentation effort.

5.3 Threats to Validity

Evaluating the comment size is not the best metric for documentation effort. Nevertheless, it is a good indicator. Even if comments contain a high amount of redundancy and, thus, can be created easily by copying certain parts, they, in turn, become harder to maintain. Moreover, other evaluation methods, such as user studies, would suffer from different documentation styles.

Since we documented the evaluated SPLs by ourselves, we possibly biased the results. To lower the impact of this issue on the evaluation, we created the comments for the straightforward approaches from our extended syntax and did not write them separately. This guarantees that no comment contains additional information that would affect the input size for our method or one of the straightforward approaches.

Another issue is the number and size of our case studies. Since there are currently only two SPLs that are documented with our extended syntax, only these two SPLs were evaluated, which could possibly lead to biased results. However, the results we got from the evaluation are similar for both SPLs and confirm our expectations.

6. RELATED WORK

As mentioned in Section 2, our method only considers source-code documentation. This is different from end-user documentation, such as manuals, which explain the functionality of a program and how to use it correctly. For SPLs there are already solutions to generate end-user documentation. DocLine is a tool to generate tailor-made user documentation by treating user documentation as SPLs [9]. It uses the XML-based documentation reuse language (DRL) that is able to reuse adapted text modules. In addition, Rabiser et al. introduced a general approach to generate arbitrary documentation using the tool suite DOPLER (decision-oriented product line engineering) [13].

Documentation is considered an informal type of specification. In contrast, there are also formal specification approaches, such as method contracts. These define the behavior and allowed return values of a method for certain inputs and are used to check the correctness of programs. Thüm et al. adapted contracts for SPLs [16]. Their work enables contract-based analyses of feature interactions, which ensures correct behavior of the implementation in all products.

7. CONCLUSION AND FUTURE WORK

Source-code documentation is a vital aspect of software engineering and thus, should be part of new software development techniques like FOP. At present, documentation tools, such as Javadoc, do not work well with FOP. Therefore, we presented four different documentation types for SPLs that provide information for several use cases, such as implementing, maintaining, using, or reusing SPLs. Using these documentation types, developers working with SPLs are provided with tailor-made documentation for each use case. However, at the moment, we lack methods to create such documentation. Therefore, we introduced a new way to structure documentation comments and designed a generation algorithm that is able to generate all four documentation types from the same source-code comments. Based on the prototypical implementation of our algorithm, we evaluated our method in terms of documentation effort for developers. The evaluation results showed that our new structure introduces little overhead and redundancy and is superior to most straightforward approaches.

In future work, we would like to investigate documentation methods for other SPL implementation techniques (e.g., pre-processors or aspect-oriented programming). We are also interested in supporting developers in the documentation process with our extended syntax by providing methods that help identify feature-independent parts of the documentation and determine proper priorities.

ACKNOWLEDGMENTS This work is partially funded by BMBF grant 01IS14017B and is based on the main author's bachelor thesis (only available in German).

8. REFERENCES

[1] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer, 2013.

[2] S. Apel, C. Kästner, and C. Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *TSE*, 39(1):63–79, 2013.

[3] S. Apel and C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *SC*, pages 20–35, 2008.

[4] D. Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *ICSE*, pages 702–703. IEEE Computer Society, 2004.

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *TSE*, 30(6):355–371, 2004.

[6] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2001.

[7] G. Holl, P. Grünbacher, and R. Rabiser. A Systematic Review and an Expert Survey on Capabilities Supporting Multi Product Lines. *IST*, 54(8):828–852, 2012.

[8] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[9] D. V. Koznov and K. Y. Romanovsky. DocLine: A Method for Software Product Lines Documentation Development. *PCS*, 34(4):216–224, 2008.

[10] C. W. Krueger. Software Reuse. *CSUR*, 24(2):131–183, 1992.

[11] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques.* Springer, 2005.

[12] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP*, pages 419–443. Springer, 1997.

[13] R. Rabiser, W. Heider, C. Elsner, M. Lehofer, P. Grünbacher, and C. Schwanninger. A Flexible Approach for Generating Product-Specific Documents in Product Lines. In *SPLC*, pages 47–61. Springer, 2010.

[14] R. Schröter, N. Siegmund, T. Thüm, and G. Saake. Feature-context interfaces: tailored programming interfaces for software product lines. In *SPLC*, pages 102–111. ACM, 2014.

[15] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *SCP*, 79(0):70–85, 2014.

[16] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, and G. Saake. Applying Design by Contract to Feature-Oriented Programming. In *FASE*, pages 255–269. Springer, 2012.