Embedded Annotations Specification

Version 0.1

The embedded annotations design serves the purpose to describe how to document software feature locations close to the source code artifacts level. In general, there are two ways to locate features in a software product: First, the "lazy" approach where to locate them when needed and second, the "eager" approach to document feature location while development. The here chosen approach is the "eager" one, which can be either reached with external tooling or as used here, to document the feature locations directly in source code or specialized files close to it. Embedded annotations offer the benefit - to externally documented feature locations - that they evolve naturally with the source code itself (Ji et al., 2015) and while cloning of source code in Clone&Own actions, allow propagating changes over software variants. Embedded annotations cover either blocks or specific lines of source code or file system resources. With this flexible approach, it is possible to annotate projects on a system level, e.g. to benefit from object-oriented programming, folders, and files reflect the internal structure, and at the same time to annotate line-specific feature relations. The way how these annotations work is independent of any project programming languages and can be also applied to non-source code files, such as e.g. configuration or binary files.
Embedded annotations fulfill the purpose of traceability and neither required central management nor to be pre-defined.

Features play in modern software development a central role. In general agile software development focuses on customer functionality and features, whereby the method "Feature Driven Development (FDD)" takes a special position and puts the feature as the center of every decision and following the agile manifesto. (Wikimedia, 2019)

Locating features in source code is an important work for software developers (Entekhabi et al., 2019). The benefit to document features is seen in most cases only in the long run or with high coverage of the source code but can reduce feature location costs significant (Ji et al., 2015). Currently several slightly different approaches exist to write feature locations in project artifacts, known as embedded annotations (Ji et al., 2015; Andam et al., 2017; Entekhabi et al., 2019; Krüger et al., 2019). The here proposed notion unifies these approaches and allows the implementation of reference software libraries to it.

Contents

1 Formal Definition of Embedded Annotations

Embedded Annotations Terminologies

The design for embedded annotations requires some special terminologies:

Feature A distinct functionality or attribute of a software product, usually expressed in functional or non-functional requirements.[1]

Feature Reference Reference to a concrete feature in the feature model.

Annotated Scope Artifacts associated with one or more features.

Annotation Marker Keyword to open/close the annotated scope for one or more feature references.

Annotation In source code: Feature Marker including all its feature references.

Feature Model A feature hierarchy model, describing feature names and their hierarchy in textual form.

Embedded Annotations Meta-Model

The Meta-Model shown in Figure 1 shows the different attributes, relations and constrains of the embedded annotation notion.

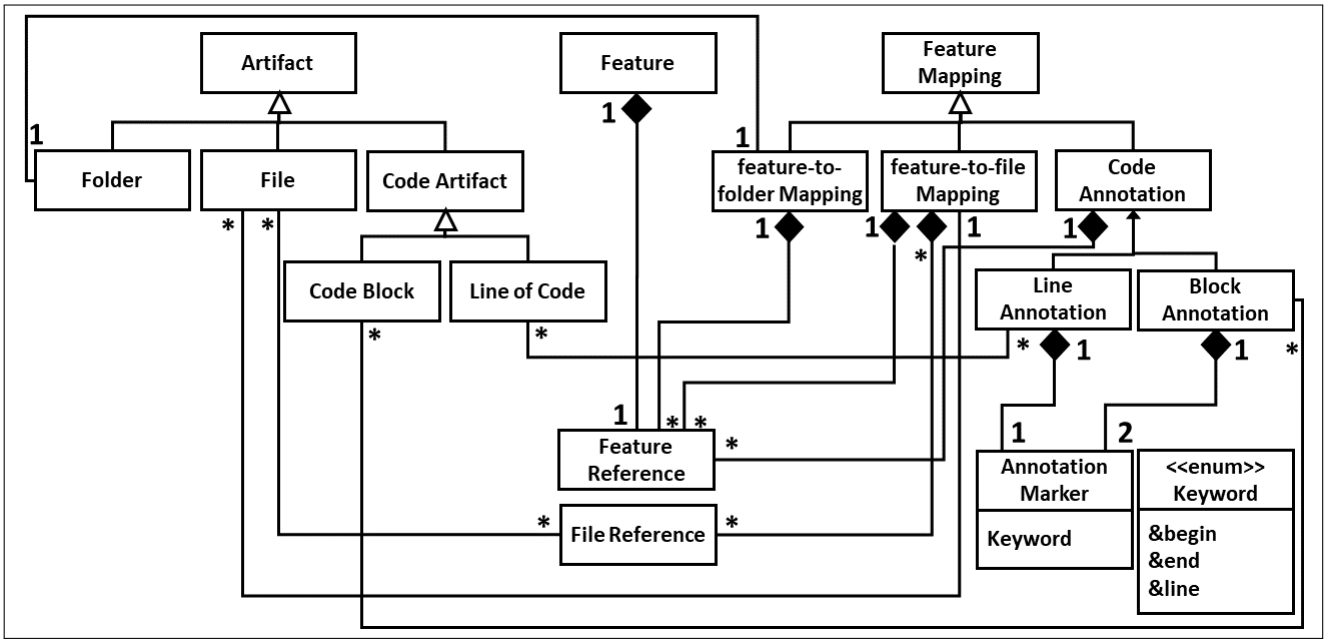

Figure 1: Meta-Model for Embedded Annotations

The Meta-Model for embedded annotations contains 16 attributes and 22 relations. The elements used to mark source code artifacts with embedded annotation belong to the attribute type *Artifact* and derive into *Folder*, *File* and *Code Artifact*. *Code Artifacts* thereby derive into

[1]Detailed analysis about feature definition in Krüger et al. (2019)

Code Block and *Line of Code* annotations. The different types of *Artifact* have all a many-to-many relationship to their feature-mapping counterparts. *File* has thereby the specialty that it contains of a feature-mapping and a *File Reference*.
A concrete *Feature* is represented by a *Feature Reference* which in the following can be used for *Feature to Folder Mapping*, *Feature to File Mapping* or *Code Annotation*; all of them from the type *Feature Mapping*. A concrete *Feature Reference* is used in a concrete mapping, but a mapping might consist of multiple *Feature References*. *Feature References* in *Code Annotations* are either *Block Annotations* or *Line Annotations*. One *Block Annotations* consist of exactly two *Annotations Marker*: "Begin" and "End". One *Line Annotation* consist of exactly one *Annotation Marker*: "Line".

Embedded Annotations Level System

The definition of embedded annotations is split into two levels. This serves the purpose to have the appropriate level of expressiveness for different purposes.

Level 1 Begin-, End- and Line-annotations, annotation identifier, Least-Partially-Qualified name, Simple Hierarchy Model, feature-to-file mapping and feature-to-folder mapping

Level 2 Level 1 + logical operator expressions, Full Hierarchy Model

Keywords

Keywords are reserved words for the usage of embedded annotations and can not be used as annotation names. Note that the limitation is not on a combination of keywords and other words, e.g. the keyword "line" in a concatenation such as "DatabaseLineReading" won't be treated by the parser as a keyword.
Keyword-List:

- &begin
- &end
- &line
- &file

2 Feature Hierarchy Model

The Feature Hierarchy Model defines the available features and their hierarchy relations in a textual format. The feature hierarchy model serves to model features and organized them in a hierarchical structure to keep an understanding of them. This model needs to be maintained by the developers themselves as they have the deepest domain knowledge. This work can be supported by SW-Architects or domain experts.
The syntax is inspired by the Clafer modeling language[2]. Feature models allow very detailed descriptions of feature hierarchy and relations in-between. For the purpose of feature modeling, a subset of these options is sufficient and presented in the following as "Simple Hierarchy Model". The full range of feature models is touched in the "Full Hierarchy Model".

Simple Hierarchy Model The simple hierarchy model covers the feature references and their hierarchy with one tab insertion per level. Each feature reference is listed as an indepen-

[2] `https://github.com/gsdlab/clafer/blob/master/doc/clafer.pdf`

dent line. For the simple hierarchy model, the hierarchy-file must be stored at the root node of the project and is exclusive per project.
Example:

```
ProjectName
    FeatureA
        FeatureA1
        FeatureA2
    FeatureB
        FeatureB1
```

Full Hierarchy Model The Full Hierarchy Model supports all language elements of feature hierarchy models. Each feature is listed as an independent line and constraints such as annotations relation, e.g. xor as mutually exclusive selection between annotations, or to mark an annotation with "?" as optional are possible. The full hierarchy model covers extended capabilities, as defined by the Clafer language, and includes also feature inheritance and nesting.
Example:

```
ProjectName
    FeatureA ?
        xor FeatureA1
        FeatureA2
    FeatureB
        FeatureB1 ?
        FeatureB2
```

In concrete implementations, the feature hierarchy file name could be _.cfr or similar as defined for this project.

3 Feature Reference Names

Inside the feature hierarchy model, features with the same name may appear twice or more often. To reference features uniquely the individual feature is pre-extended by its ancestor till the combined feature reference is unique. This technique is called Least-partially-qualified name, short LPQ.

Example:

```
CoffeeShop
    Coffee
        Sugar
        Milk
    Tea
        BlackTea
            Milk
```

Features as "Milk" are appearing twice in the overall model, the individual entities can be addressed by "Coffee::Milk" and "BlackTea::Milk".
In contrast, the fully-qualified name is much longer and more likely to change as compared to the least-partially-qualified name when the feature model evolves. In case an annotation appears only once, its LPQ is identical to its name, e.g. "Sugar". The separation of individual annotations to their ancestors is shown via the "::" characters. Approach from Andam et al. (2017).

4 Annotation Listing

Annotation identifiers are the concrete embedded annotations or features used in the source code. They are individual or combined words, without spaces or punctuation marks.
The usage of multiple annotation identifiers together is possible. In concrete implementations, the separator of annotations could be a comma, space-character, or similar as defined for this project.

The following syntax applies for the annotations listing:

```
Annotation_1, Annotation_2 [, Annotation_n]
```

The conjunction of multiple annotations causes the mapping of the marked source code to ALL listed annotations in the same way. I.e. the order of the given annotations is independent.

5 Feature expression logic

Besides mapping source code, files, and folders to features or a list of features, it might be required to map code to combinations of features, written in boolean expressions.

AND-Operator

The marked code part is considered to all given annotations individually. Comparable with multiple begin/end markers.
In concrete implementations the logical operator between individual annotations could be an “AND”, “&&” or similar as defined for this project.
The following syntax applies for combining multiple annotations in one identifier :

```
Annotation_1 AND Annotation_2 [AND Annotation_n]
```

Alternatively with symbolic characters

```
Annotation_1 && Annotation_2 [&& Annotation_n]
```

OR-Operator

The marked code part is considered to all given annotations individually. Comparable with multiple begin/end markers.
In concrete implementations the logical operator between individual annotations could be an “OR”, “||” or similar as defined for this project.
The following syntax applies for combining multiple annotations in one identifier :

```
Annotation_1 OR Annotation_2 [OR Annotation_n]
```

Alternatively with symbolic characters

```
Annotation_1 || Annotation_2 [|| Annotation_n]
```

NOT-Operator

The marked code part is considered to all annotations individually, except the given one.
In concrete implementations the logical operator for an annotation negation could be an “NOT”, “!” or similar as defined for this project.
The following syntax applies for annotation negation in one identifier :

```
NOT Annotation_1
```

Alternatively with symbolic characters

```
! Annotation_1
```

Order of operators

Annotations logic operators may appear in mixed mode. The general precedence rules of boolean expressions apply in this case.

6 Annotation Markers

There are three kinds of annotation markers: &begin, &end, and &line, each with specific purposes and syntax. Annotation markers are escaped through the programming language specific comment characters, such as e.g. "//" or "#". This avoids unwanted side effects for the project execution.

6.1 The begin-marker

In concrete implementations, this could be #ifdef, &begin, or a similar expression defined for this project.

The following syntax applies for the begin-marker:

```
   //&begin[ <parameter> ] <comment> <cr> /*<cr> carriage return*/
                                          /*is a newline symbol */
```

This marker considers the following consecutive lines of text to be part of this identifier. Identifiers can be defined in a hierarchy feature model, but must not. A begin-marker must be closed by an end-marker.

6.2 The end-marker

In concrete implementations, this could be #endif, &end, or a similar expression defined for this project.

The following syntax applies for the end-marker:

```
    //&end[ <parameter> ] <comment> <cr>
```

This marker ends the scope of the begin-marker of the given identifier. Identifiers can be defined in a hierarchy feature model, but must not. A end-marker must have a pre begin-marker.

6.3 The line-marker

In concrete implementations, this could be &line or a similar expression defined for this project. The line-marker is a convenient way to use a &begin- and &end-marker for a single line and can be substituted by them.

The following syntax applies for the line-marker:

```
    any source code //&line[ <parameter> ] <comment> <cr>
```

This marker considers exclusively its own line of text to be part of this identifier. If this line is a class or method, still only the annotated line is considered as part of this identifier.

6.4 Interleaving of Annotation Markers

Annotation markers appear in the simple case independent of each other and have no cross-cutting relation. That this is unlikely in a practical context is shown by following Listing 1. Here the annotation "Codecs" is surrounded by "RequestCoins" and "BIP21" is inside "Codecs".

Potential interleaving situations are e.g. with overlapping annotation scopes ot with same begin-multiple end or vice versa.

Overlapping annotation scopes Between different &begin- and &end-markers as well with the &line-markers the marked scope for a certain annotation might overlap. A full overlapping (left), as well as a partial overlapping (right), are possible:

```
1 //begin[FeatureA]
2 ... (code FeatureA)
3 //begin[FeatureB]
4 ... (code FeatureA, FeatureB)
5 //end[FeatureB]
6 ... (code FeatureA)
7 //end[FeatureA]
```

```
1 //begin[FeatureA]
2 ... (code FeatureA)
3 //begin[FeatureB]
4 ... (code FeatureA, FeatureB)
5 //end[FeatureA]
6 ... (code FeatureB)
7 //end[FeatureB]
```

This situation might happen also with three and more annotation markers.

Same begin-multiple end or vice versa With the possibility of &begin and &end to support multiple feature references at the same time, there is the option to start multiple features within one &begin- and end them with individual &end-markers - or vice versa.

```
1 //begin[FeatureA, FeatureB]
2 ... (code FeatureA, FeatureB)
3 //end[FeatureA]
4 ... (code FeatureB)
5 //end[FeatureB]
```

```
1 //begin[FeatureA]
2 ... (code FeatureA)
3 //begin[FeatureB]
4 ... (code FeatureA, FeatureB)
5 //end[FeatureA, FeatureB]
```

This situation might happen also with three or more feature references.

Greater flexibility than IFDEFs The approach of interleaving of annotation markers provides greater flexibility to mark features than the notion of IFDEFs. IFDEFs set for their scope a concrete set of features and close them always all together. Usage of inner or simultaneous opening with individual closing is not possible as the IFDEF-precompiler cuts out the code parts and the inner parts are lost.

Example 1: Same begin with multiple ends

```
1 #define FEATURE_FIRST
2 #define FEATURE_SECOND
3
4 #ifdef FEATURE_FIRST &&
      FEATURE_SECOND
5 ...
6 #endif /* FEATURE_FIRST */
7 ...
8 #endif /* FEATURE_SECOND */
```

This setup causes a compiler error "#endif without #if" as #ifdef allows exactly one endpoint about its scope. I.e. multiple-ends or multiple starts are technically not possible.

Example 2: Partially overlapping feature code (lines 4-8 and lines 6-10)

```
1 #define FEATURE_FIRST
2 #define FEATURE_SECOND
3
4 #ifdef FEATURE_FIRST
5 ...
6 #ifdef FEATURE_SECOND
7 ...
8 #endif /* FEATURE_FIRST */
9 ...
10 #endif /* FEATURE_SECOND */
```

Disabling "FEATURE_SECOND" cause to early end of scope at first found #endif:

```
1 #define FEATURE_FIRST
2 //#define FEATURE_SECOND
3
4 #ifdef FEATURE_FIRST
5 ...
6 #ifdef FEATURE_SECOND
7 ...
8 #endif /* FEATURE_FIRST */
9 ...
10 #endif /* FEATURE_SECOND */
```

7 Feature Mappings

Embedded annotations are mainly used to annotate source code parts. For the mapping of hole folders and folders including their subfolders, two kinds of mappings exist: feature-to-file and feature-to-folder.

7.1 Feature-to-code mapping

The feature-to-code mapping serves to link specific blocks and lines of code to one or more features. The parts of the source code which are mapped to a certain feature are called annotation scopes. An annotation scope is surrounded by annotation markers (see chapter 6) and contains at least one feature reference (see chapter 3).

7.2 Feature-to-file mapping

The feature-to-file mapping is a specialized file to map one or more file(s) and its/ their content to one or more feature references. All content of the linked file is considered fully to be part of the given feature references. The mapping file must be stored in the same folder as the source code files and covers only the file in this folder. This is especially helpful to map files that don't contain source code, such as binary or generated files. The feature-to-file mapping is exclusive for files and folders to be mapped to features by the feature-to-folder mapping - even when some operating systems handle files and folders identical (e.g. Linux).
In case the file name contains spaces or other special characters, the file name can be escaped with leading and ending quotation marks, e.g. "database config.dab".

For projects with little applicability, or if the files for feature-to-file mapping wants to be avoided, the same effect of mapping the complete content of a file can be reached with a "begin"-annotation marker at the beginning of the file and its "end"-annotation marker counterpart at the end of the file.

The following syntax applies for the mapping file:

Alternative 1

```
File_a (File_b ...) <cr>
Feature_1 (Feature_2 ...) <cr>
File_x (File_y ...) <cr>
Feature_n (Feature_m ...) <cr>
<eof>
```

Alternative 2

```
File_a (File_b ...) <cr>
Feature_1 <cr>
Feature_2 <cr>
...
File_x (File_y ...) <cr>
Feature_n <cr>
Feature_m <cr>
...
<eof>
```

In concrete implementations, this filename could be _.feature-file or similar as defined for this project. To avoid that the file is hidden by the development IDE or file explorer the leading underscore is recommended.

Alternative 3 Instead of handling the feature-to-file mapping in an own file, the same result can be reached with the file-marker which should be placed at the file-header, best initial line.

The following syntax applies for the file-marker:

```
    //&file[ <parameter> ] <comment> <cr>
```

This marker considers the hole file to the given features in the parameter. The benefit is the independence of the specialized file and that with a copy/move the feature information is automatically considered. The drawback is that a feature mapping change is not independent possible of the file, e.g. using in another project or scope, and technically not possible for binary files.

The shown "Alternative 2" is dismissed for the implementation part. The reason for this is the challenge to handle files without a file extension, e.g. a file named "database". The tool won't be able to detect if this is a file or feature. Therefore a safe mapping from files and features is not possible with this approach.

The shown "Alternative 3" is not considered in the implementation part. The reason for this is the late appearance of this option and that the functionality is available with the "Alternative 1".

7.3 Feature-to-folder mapping

The purpose of this file is to map complete folders and their content to one or more feature references. The mapping of feature references to folders allows linking specific features to the folder, including all its sub-folders and files. With this, the mapping of complete folder structures to features is possible and may substitute the feature-to-file mapping. The mapping file is located on the top level inside the to be annotated folder.
This way of the feature-to-folder has two main benefits. Firstly, the mapping takes care that always all containing artifacts are linked to the feature(s) and provides, therefore, more stability on a folder base than feature-to-file where the developer would need to maintain the file list. And secondly, the location inside the folder is more stable against renaming or moving the folder than the feature-to-file mapping.

The following syntax applies for the mapping file:

Alternative 1

```
<LPQ> <cr>
<LPQ> <cr>
<eof>
```

Example:

```
Feature_1 <cr>
Feature_n <cr>
<eof>
```

Alternative 2

```
<LPQ> , <LPQ> <cr>
<eof>
```

Example:

```
Feature_1 , Feature_n <cr>
<eof>
```

In concrete implementations, this filename could be _.feature-folder or similar as defined for this project. It is stored inside the folder it annotates to resists better changes of the folder, such as renaming or moving.

8 Embedded Annotation Examples

8.1 Annotation Code Examples

To illustrate the defined begin-, end- and line-markers, two real-world examples shall illustrate the usage.

```
224 public static PaymentIntent fromBitcoinUri(final BitcoinURI bitcoinUri) {
225     final Address address = bitcoinUri.getAddress();
226     final Output[] outputs = address != null ? buildSimplePayTo(bitcoinUri.getAmount(), address
        ) : null;
227
228     final String bluetoothMac = (String) bitcoinUri.getParameterByName(Bluetooth.MAC_URI_PARAM)
        ; //&line[Bluetooth]
229
230     //&begin[RequestCoins]
231     final String paymentRequestHashStr = (String) bitcoinUri.getParameterByName("h");
232     final byte[] paymentRequestHash = paymentRequestHashStr != null ? base64UrlDecode(
        paymentRequestHashStr) : null;
233
234     //&begin[Codecs]
235     return new PaymentIntent(PaymentIntent.Standard.BIP21, null, null, outputs, bitcoinUri.
        getLabel(), //&line[BIP21]
236     //&end[Codecs]
237             bluetoothMac != null ? "bt:" + bluetoothMac : null, null, bitcoinUri.
        getPaymentRequestUrl(),
238             paymentRequestHash);
239     //&end[RequestCoins]
240 }
```

Listing 1: Code example embedded annotations: Bitcoin-Wallet, class PaymentIntent. Adjusted from (Krüger et al., 2019).

Explanation of used markers and embedded annotations inside Listing 1:

- Line 228 "//&line[Bluetooth]" belongs to the feature Bluetooh and the line-marker matches exclusive line 228 to the feature Bluetooth.
- Line 230 "//&begin[RequestCoins]" belongs to the feature RequestCoins and the begin-marker maps lines 230 and following to the feature RequestCoins. This scope is ended with the "//&end[RequestCoins]" at line 239.
- Line 234 "//&begin[Codecs]" belongs to the feature Codecs and the begin-marker maps lines 234 and following to the feature Codecs. This scope is ended with the "//&end[Codecs]" at line 236.
- Line 235 "//&line[BIP21]" belongs to the feature BIP21 and the line-marker matches exclusive line 235 to the feature BIP21.

```
115 public class BlockchainService extends LifecycleService {
116     private WalletApplication application;
117     private Configuration config;
118     private AddressBookDao addressBookDao;  //&line[AddressBook]
        ...
139     private long serviceCreatedAt;
140     private boolean resetBlockchainOnShutdown = false;  //&line[ResetBlockChain]
        ...
157     //&begin[ResetBlockChain]
158     private static final String ACTION_RESET_BLOCKCHAIN = BlockchainService.class.getPackage()
        .getName()
159             + ".reset_blockchain";
160     //&end[ResetBlockChain]
```

```
161     //&begin[BlockchainSync]
162     private static final String ACTION_BROADCAST_TRANSACTION = BlockchainService.class.
        getPackage().getName()
163             + ".broadcast_transaction";
164     private static final String ACTION_BROADCAST_TRANSACTION_HASH = "hash";
165     //&end[BlockchainSync]
166
167     private static final Logger log = LoggerFactory.getLogger(BlockchainService.class);
```

Listing 2: Code example embedded annotations: Bitcoin-Wallet, class BlockchainService. Adjusted from (Krüger et al., 2019).

Explanation of used markers and embedded annotations inside Listing 2:

- Line 115 "//&line[AddressBook]" belongs to the feature AddressBook and the line-marker matches exclusive line 115 to the feature Bluetooth.
- Line 140 "//&line[ResetBlockChain]" belongs to the feature ResetBlockChain and the line-marker matches exclusive line 140 to the feature Bluetooth.
- Line 157 "//&begin[ResetBlockChain]" belongs to the feature ResetBlockChain and the begin-marker maps lines 157 and following to the feature ResetBlockChain. This scope is ended with the "//&end[ResetBlockChain]" at line 160.
- Line 161 "//&begin[BlockchainSync]" belongs to the feature BlockchainSync and the begin-marker maps lines 161 and following to the feature BlockchainSync. This scope is ended with the "//&end[BlockchainSync]" at line 165.

8.2 File Mapping Examples

```
1  ProjectRoot
2  |-- feature-model.cfr
3  |-- src
4     |-- de
5        |-- schildbach
6           |-- wallet
7              |-- data
8              |-- offline
9              |-- service
10             |-- ui
11             |-- util
12             |-- _.feature-file
13             |-- _.feature-folder
14             |-- Configuration.java
15             |-- Constants.java
16             |-- Logging.java
17             |-- WalletApplication.java
18             ...
```

Listing 3: Code example mapping files: Bitcoin-Wallet, folder structure. Adjusted from (Krüger et al., 2019).

```
1  BitcoinWallet
2      OwnName
3      Bluetooth
```

Listing 4: Code example mapping files: Bitcoin-Wallet, feature-model.cfr. Adjusted from (Krüger et al., 2019).

```
1  Configuration.java
2  OwnName
```

Listing 5: Code example mapping files: Bitcoin-Wallet, _.feature-file. Adjusted from (Krüger et al., 2019).

```
1  Main
```

Listing 6: Code example mapping files: Bitcoin-Wallet, _.feature-folder. Adjusted from (Krüger et al., 2019).

Explanation of used mappings inside Listings 4 till 6. Listing 3 shows the folder structure for the mapping files.

- Listing 4 shows the feature hierarchy model (Folder structure line 2) with:

  - Line 1 the project name and root node for the hierarchy.
  - Line 2 and 3 the features “OwnName” and “Bluetooth”, both in the first hierarchy level. Even when more features are used by the project, they are not required to be listed here.

- Listing 5 shows the feature-to-file mapping (Folder structure line 12) of the file “Configuration.java” (Folder structure line 14) to feature “OwnName”.
- Listing 6 shows the feature-to-folder mapping (Folder structure line 13) and therefore inside folder “wallet”. This means that the folder wallet and all its content is mapped to the feature “Main”.

Fully annotated projects have been created by Krüger et al. (2019) and can be found at `https://bitbucket.org/rhebig/jss2018/`.

References

Andam, Berima et al. (2017). "FLOrIDA: Feature LOcatIon DAshboard for Extracting and Visualizing Feature Traces". In: *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-Intensive Systems*. VAMOS '17. Eindhoven, Netherlands: Association for Computing Machinery, pp. 100–107. ISBN: 9781450348119. DOI: 10.1145/3023956.3023967. URL: https://doi.org/10.1145/3023956.3023967.

Entekhabi, Sina et al. (Sept. 2019). "Visualization of Feature Locations with the Tool FeatureDashboard". In: pp. 1–4. ISBN: 978-1-4503-6668-7. DOI: 10.1145/3307630.3342392.

Ji, Wenbin et al. (2015). "Maintaining Feature Traceability with Embedded Annotations". In: *Proceedings of the 19th International Conference on Software Product Line*. SPLC '15. Nashville, Tennessee: Association for Computing Machinery, pp. 61–70. ISBN: 9781450336130. DOI: 10.1145/2791060.2791107. URL: https://doi.org/10.1145/2791060.2791107.

Krüger, Jacob et al. (2019). "Where is my feature and what is it about? A case study on recovering feature facets". In: *Journal of Systems & Software* 152, pp. 239–253. ISSN: 01641212. DOI: 10.1016/j.jss.2019.01.057. URL: http://search.ebscohost.com/login.aspx?direct=true&AuthType=sso&db=buh&AN=135661128&site=eds-live&scope=site&custid=s3911979&authtype=sso&group=main&profile=eds.

Wikimedia, Foundation Inc. (2019). *Feature-driven development*. https://en.wikipedia.org/wiki/Feature-driven_development. Accessed: 2020-05-15.

A Supported Embedded Annotations to Engine Versions

Specification Element	Version 0.1
Feature Reference Names in LPQ	X
Annotations Listing	X
Feature expression logic (AND, OR, NOT)	-
Feature Hierarchy Model	
Simple Hierarchy Model	X
Full Hierarchy Model	-
Feature Model File below root folder	X
Feature Model File location flexible	-
Annotation Markers	
&begin	X
&end	X
&line	X
&file	-
Feature Mappings	
Feature-to-code mapping	X
Feature-to-file mapping - Alternative 1	X
Feature-to-file mapping - Alternative 2	-
Feature-to-file mapping - Alternative 3	-
Feature-to-folder mapping - Alternative 1	X
Feature-to-folder mapping - Alternative 2	X

Table 1: Embedded Annotations Available (X) and Not Available (-) in Specific Engine Versions.