Embedded Annotations Specification

The embedded annotations specification serves the purpose to document software feature locations close to the source code artifacts level. In general there are two ways to locate features in your product: First, the "lazy" approach where you 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 in specialized files close to it. Embedded annotations offer the benefit - to externally documented feature locations - that they are evolve naturally with the source code itself (Ji et al., 2015) and while cloning of source code in Clone&Own actions, allow to propagate changes over software variants. Embedded annotations cover either blocks or specific lines of source code or file system resources.
Embedded annotations fulfill the purpose of traceability and neither required a central management nor to be pre-defined.
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 your source code, but can reduce feature location costs significant (Ji et al., 2015). Currently several slightly different approaches exist to write feature locations in your 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 specification unifies these approaches and allows the implementation of reference software libraries to it.

Contents

1 Formal definition embedded annotations

Embedded Annotations Terminologies

This specification is using some special terminologies:

Feature A distinct functional or non-functional attribute of a software product.[1]

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

Annotated Scope Artifacts associated to 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 simple textual feature model, using a subset of the Clafer modeling language (i.e. feature names and hierarchy).

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 Clafer Hierarchy, feature-to-file mapping and feature-to-folder mapping

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

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 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 keyword.

Keyword-List:

- &begin
- &end
- &line

2 Clafer Feature Model

The Clafer Feature Model defines the available features and their hierarchy relations in a textual format. The Clafer syntax is described in more detail in the Clafer documentation[2].

Simple Clafer Hierarchy The simple Clafer hierarchy covers the feature references and their hierarchy with one tab insertion per level. Each feature reference is listed as independent line. For the simple Clafer hierarchy, the Clafer-file must be stored at the root node of the

[1] More about what a features is in Krüger et al. (2019)

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

project and is exclusive per project.
Example:

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

Full Clafer Hierarchy The Full Clafer Hierarchy supports all language elements of the Clafer language, including Clafer inheritance and nesting, in combination with embedded annotations. Each annotation is listed as independent line and constrains such as annotations relation, e.g. xor as mutually exclusive selection between annotations, or to mark an annotation with "?" as optional are possible.
Example:

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

In concrete implementations the Clafer file name could be _.cfr or similar as defined for this scope.

3 Feature References

Inside the Clafer Feature 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 scope.

The following syntax applies for the annotations listing:

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

The conjunction of multiple annotations cause 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

Beside mapping source code 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. Just as multiple begin/end markers.
In concrete implementations the logical operator between individual annotations could be an "AND", "&&" or similar as defined for this scope.
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. Just as multiple begin/end markers.
In concrete implementations the logical operator between individual annotations could be an "OR", "||" or similar as defined for this scope.
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 scope.
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 scope.

The following syntax applies for the begin-marker:

```
  //&begin[ <parameter> ] <comment> <newline>
```

This marker considers the following consecutive lines of text to be part of this identifier. Identifiers can be defined in a textual Clafer 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 scope.

The following syntax applies for the end-marker:

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

This marker ends the scope of the begin-marker of the given identifier. Identifiers can be defined in a textual Clafer 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 scope. 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> <newline>
```

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".

Further interleaving situations are e.g.

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) is possible:

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

```
1 //begin[FeatureA]
2 ...
3 //begin[FeatureB]
4 ...
5 //end[FeatureA]
6 ...
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 and them with individual &end-markers - or vice versa.

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

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

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

7 Feature Mappings

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

7.1 The 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 The 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 which don't contain source code, such as binary or generated files. The feature-to-file mapping is exclusive for files, folders need to be mapped 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".

The following syntax applies for the mapping file:

```
file_a (file_b) <newline>
Reference_1 (Reference_2) <newline>
file_x (file_y) <newline>
Reference_n (Reference_m) <newline>
<End-of-file>
```

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

7.3 The 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 allow to link specific features to the parent folder of this file, 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 feature-to-folder 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.

The following syntax applies for the mapping file:

Alternative 1

```
<LPQ> <newline>
<LPQ> <newline>
<End-of-file>
```

Example:

```
Reference_1 <newline>
Reference_n <newline>
<End-of-file>
```

Alternative 2

```
<LPQ> <LPQ> <newline>
<End-of-file>
```

Example:

```
Reference_1 Reference_n <newline>
<End-of-file>
```

In concrete implementations this file name could be _.feature-folder or similar as defined for this scope.

7.4 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 231 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 235 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.
- The annotations scopes RequestCoins (line 230-239) and Codecs (line 234-

```
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 158 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 162 and following to the feature BlockchainSync. This scope is ended with the "//&end[BlockchainSync]" at line 165.

7.5 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  OwnName
2  Configuration.java
```

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 Clafer feature 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 feature “OwnName” to the file “Configuration.java” (Folder structure line 14).
- 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”.

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.