An Empirical Investigation of Code Comments Evolution and Practices: the Case of the Pharo Environment

I. Template-inspired categories

1) **Intent**: The most frequent category *Intent* describes the purpose of the class and answers the question 'Who am I?'. In Pharo, developers usually explain the intent of the class in the first line of a comment, using the pattern `I represent`. Other often used variations of this pattern are `I am`, `This is`, `*Classname is`, `A *Classname is` *etc.*
A couple of examples are shown in Listing 1, and Listing 2.

```
I represent a message to be scheduled by the
    ↪ WorldState.
```

Listing 1: Intent of the MorphicAlarm class

```
I'm a TextMorph that must be embedded in a
    ↪ PluggableTextMorph.
```

Listing 2: Intent of the TextMorphForEditView class

2) **Responsibility:** This category answers the questions 'what do I know?' and 'what do I do?'. We find in most of the cases this information is combined in the same sentence with the intent of the class, *e.g.*, in Listing 3.

```
I am a minimal user interface that shows the
    ↪ last entries in the watchpoint history..
```

Listing 3: Responsibility of the WatchpointWindow class

3) **Collaborator:** This category answers the question 'How I interact with the other classes?' We find this information is mentioned very rarely under a separate section, as asked in the template.
It is generally interweaved with other details as shown in Listing 4.

```
    I am a singleton object, subscribed to
    ↪ system events. When I listen one of those
    ↪  events, I reinitialize the KMRepository
    ↪ default instance and reload it with all
    ↪ declared keymaps.
```

Listing 4: Collaborator `KMRepository` in the KMPragmaKeymapBuilder class

4) **Public API:** When developers want to use a class, one of the important pieces of information they seek is the public API of the class and the usage of the API. We gathered this detail by answering to the question 'what are my Public API and Key messages?' We find that not all classes describe their public APIs in comments and not all public APIs of the class are mentioned. The APIs mentioned are those which are considered important according to the developer who is writing the comment. *e.g.*, FTAllItemsStrategy class have eight total messages and three of them are exposed in other classes but not all are mentioned in the comments and only one API 'realSearch' is mentioned under Public API and Key Messages section.

```
    Public API and Key Messages
    -----------------------

    - #realSearch is the method that will
    ↪ launch the search.
```

Listing 5: Key Messages `realSearch` in the FTAllItemsStrategy class

5) **Example:** This category presents examples embedded in the comments to give readers a better overview of the class with small code snippets. In studies of developers, examples to use API found to be a major learning resource [1], [2], [3]. We find that developers write code examples to instantiate the class and to use API of the class. shown in Listing 6. Sometimes examples are given in plain natural language as a simpler explanation of the concepts used in the class. shown in Listing 7.

```
    Usage examples:

    GTInspector new openInExternalWindowOn: 3.

    3 inspectInExternalWindow.
```

Listing 6: Code example in the GLMOSWindowWorldMorph class

```
    I can opened based on opening strategy (
    ↪ hover, shift + hover, or double click +
    ↪ shift).
```

Listing 7: Example in the GLMEmptyPopupBrick class in natural language

6) **Implementation Points:** This category answers the question 'What are my implementation and internal details?'. Internal details refer to the internal representation of the objects, particular implementation logic, conditions about the object state, and settings important to under-

stand the class as shown in Listing 8 and Listing 9.

```
I'm a variant of FFICalloutMethodBuilder that
    ↪ adds a call to #signalIfNotReady before
    ↪ the actual ffi call.
```

Listing 8: Implementation condition given in the LGitSafeFFICalloutMethodBuilder class

```
    If one of the entities is new or deleted the
    ↪  "nfiles" field will hold be set to 1,
    ↪ otherwise it will be set to 2.
```

Listing 9: Internal representation of the objects given in the LGitDiffDelta class

7) **Instance Variables:** When developers want to know about the state of the object they can consult the instance variables section of a comment. We gathered this section by asking the question 'what are my instance variables?'. Developers mostly document the rationale and behavior of instance variables of the class *e.g.*, name, type of object and rational of instance variables.
is given in RBCascadeNode class Listing 10.

```
Instance Variables:
messages      <SequenceableCollection of:
    ↪ RBMessageNode>    the messages
semicolons    <SequenceableCollection of:
    ↪ Integer>        positions of the ;
    ↪ between messages
```

Listing 10: Details of instance variables given in the RBCascadeNode class

II. Template non-inspired categories

There are other types of details we found in comments and are not mentioned in the template.

1) **Class references** While describing the class context, developers refer to other classes very frequently. **??** confirms this hypothesis. The category generally overlaps with Collaborator category but includes extra cases when developers refer to other classes in the class comment to explain the context of the class. The developers can refer to other classes in describing the intent of the class, warnings about the class, and examples about the class *e.g.*, the class comment of the class *SparseLargeArray* refer to the class *SparseLargeTable* (Listing 11).

```
        A version of SparseLargeTable that does
    ↪ not populate its bins until a value other
    ↪  than the default is stored.!
```

Listing 11: Class reference given in the SparseLargeArray class

We observe that developers sometimes refer to other classes by the exact name, whereas a few times developers break the long class name into separate words. Also, the classes names are not always capitalized.

2) **Warnings:** When developers want to warn their readers with some important piece of information about the class, they write warnings, alerts or general notes in their comment. We believe that warnings help readers to pay attention to common pitfalls in using a class. Sometimes developers explicitly denote warnings with a heading, whereas many times warnings are just woven into text flow without any markup.
We show an example of an explicit warning in Listing 12 and an implicit warning in Listing 13.

```
    **NOTE**
    As a workaround of bitblt bug, the actual
    ↪ Cairo surfaces, created internally is
    ↪ with 1 extra pixel higher than requested.
    ↪  This is, however completely hidden from
    ↪ users.
```

Listing 12: Explicit warning given in the AthensCairoSurface class

```
    They shouldn't be directly used and always
    ↪ be a part of a refactoring namespace -
    ↪ the model.
```

Listing 13: Implicit warning given in the RBAbstractClass class

3) **Precondition:** Before using a class or a functionality, developers want to be informed about potential preconditions that have to be met in order to use a class. Again we find different styles of how preconditions are expressed. In some cases preconditions are grouped in a separate header, sometimes they are expressed implicitly.
(see Listing 14), sometimes they are expressed implicitly (see Listing 15).

```
        Preconditions:
        - the class must exist
        - the category or package must exist
```

Listing 14: Explicitly mentioned Precondition in the RBMoveClassTransformation class

```
        It will only execute if the presentation
    ↪  is directly rendered in a window (i.e.,
    ↪ if this is the presentation to which #
    ↪ openWith: was sent)
```

Listing 15: Implicitly mentioned Precondition in the GLMWindowRequest class

4) **Dependencies:** This category shows the dependency of the class on other classes or components. The dependency can be a class, a method in other class or other packages *e.g.*, in Listing 16 the class shows the dependency on cario graphics library. This can be a piece of important information if developers want to extend the class.

```
i am a concrete implementation of Athens
↪ surface which using cairo graphics
↪ library for rendering.
```

Listing 16: Dependency given in the AthensCairoSurface class

5) **Reference to other resources:** Developers mention external resources in the class comment to refer the reader to other resources and gain a better view and related information of the class.
as shown in Listing 17.

```
Relationship to SketchMorph:
↪ ImageMorph should be favored over
↪ SketchMorph, a parallel, legacy class --
↪ see the Swiki FAQ for details ( http://
↪ minnow.cc.gatech.edu/squeak/1372 ).
```

Listing 17: Reference to external resources given in the ImageMorph class

The resources can be internal in the system, a class, configuration, other class comment, methods or external like a web link to other resources to give more insights on the class working *e.g.*, in Listing 18, developer points to other class *AbstractWidgetPresenter* and also method *exampleRegisteredColor* in *ExampleListPresenter* to gain more insights on the class working. We group the sentences where developers explicitly refer readers to other resources. We observe that most of the times developers use *see* and *look* to refer to other resources.

```
See AbstractWidgetPresenter.

You can also have a look at
↪ ExampleListPresenter >>
↪ exampleRegisteredColor and
↪ ListSelectionPresenter for more examples.
```

Listing 18: Reference to internal resources given in the ListPresenter class

6) **Discourse:** Interestingly, developers in Pharo use comments not just to document the class details but also to communicate to readers to inform them about a few details in an informal manner, *e.g.*, indicating the need of optimizing the class in Listing 19. We also found cases where developers use conversational language to express their thoughts about the class. One such instance is given in Listing 20.

```
FLPointCluster is an optional class that
↪ optimizes Point instances, since there
↪ are a lot of instances in the system, it
↪ makes sense to optimize them.
```

Listing 19: A discourse present in the FLPointCluster class

```
I hold onto my two subclasses
↪ OCASTTranslatorForValue for generating
↪ instructions for effect and value, and
↪ OCASTTranslatorForEffect for generating
↪ instructions for effect only.
 Which one to use depends on the AST nodes
↪ and whether the code will only be executed (
↪ for effect only) or if the value is used
↪ afterwards (for value).
```

Listing 23: Subclasses explanation in the OCASTTranslator class

```
(Even though the name doesn't look so, it is
↪  what it is.)
```

Listing 20: A discourse present in the CNGBTextConverter class

7) **Recommendation:** This category detects the recommendations developers give to improve the class. Such a detail can be very beneficial to other developers during the software maintenance tasks as it hints about the future direction of the changes like in Listing 21 and the future recommendation like in Listing 22 that can take place in the class.

```
Then once the experiments and a good
↪ solution is found it may be the time to
↪ remove me and to think that I'm an over
↪ engineered solution.
```

Listing 21: Future suggestion given in the RBClassModelFactory class

```
It is recommended to reuse #basicCheck:
↪ functionality in #checkClass: and #
↪ checkMethod:
```

Listing 22: Recommendation present in the RBDefineBasicCheckRule class

8) **Subclasses explanation:** A class holds not just details about itself but also about its subclasses, the intent of creating the subclasses, and when to use which subclass is also described. We found developers document such details in the root classes. For instance, in Listing 23, the class has two subclasses. The intent of creating subclasses and when to use which subclass is also described.
9) **Observations:** While working with a class, developers may observe certain behaviors and share these observations with other developers by reporting them in the class comment. We identified such information about developers' experience and grouped in the *Observation* category. We believe that this piece of information is essential in software maintainece tasks, as this experience report can help other developers in saving the time to investigate the same details. Two such examples we found in comments is shown in Listing 24, and Listing 25.

```
Observations about the transformation:
- If the variable is already assigned in
↪ this method, the transformation will add
↪ the new assignment just after it.
- Otherwise, the assignment will be
↪ positioned as the first statement of the
↪ method
```

Listing 24: Observation present in the RBAddAssignmentTransformation class

```
This is deprecated class because the
↪ original class TreeNodeModel was renamed
↪ to TreeNodePresenter. You should stop to
↪ use this class and modify your code to
↪ use TreeNodePresenter.
```

Listing 25: Observation present in the TreeNodeModel class

Further categories: Developers mention other types of details as well. These details are generally written in an informal way without using standard headers. Albeit not so frequent, these rarely used categories give interesting insights, *e.g.*, *License* to store license information of the code, *Extension* to tell about the possible extensions of the class, *Naming conventions* to record the different naming convention like acronyms they use in their code and *Coding Guideline* to describe rules to be followed. In *Coding Guideline*, developers suggest or describe the design pattern they use or should be used. Also, we observe developers write *Link* to refer to a web link for extra or detailed information, *TODO comments* to record actions to be done or remarks for themselves and for other developers. The remaining details for which we do not find a category to fit, we classify as *Other*. The category, *Other* includes the comments from other programming languages.

REFERENCES

[1] R. P. L. Buse and W. Weimer, "Synthesizing API usage examples," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 782–792. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337316

[2] J. Stylos, B. A. Myers, and Z. Yang, "Jadeite: Improving api documentation using usage information," in *CHI '09 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '09. New York, NY, USA: ACM, 2009, pp. 4429–4434.

[3] R. Holmes, R. Cottrell, R. J. Walker, and J. Denzinger, "The end-to-end use of source code examples: An exploratory study," in *2009 IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 555–558.