An Annotation Scheme to Support Analysis of Programming Activities

Sebastian Jekutsch

AG Software Engineering
Freie Universität Berlin, Germany
jekutsch@inf.fu-berlin.de

Abstract

This paper describes a framework for partitioning programming sessions – including coding, browsing, thinking, reading, testing, etc. – in programming episodes of five seconds' to a few minutes' duration. It is based on interpreting excerpts of a session via annotating activity types, properties, triggering events, and focus of attention. The set of predefined activities is grounded on about three hours of actually observed episodes. The level of abstraction is well above single keystrokes but below what is usually called programming cycle, i.e. phases of changing and testing code. The concepts of the framework are described using real world examples. The annotation scheme has been developed to aid in detecting behavioural patterns, especially for analysing defect injection episodes.

Keywords: coding scheme, ethnographic studies, actual process, programming activities, software engineering

1. Introduction

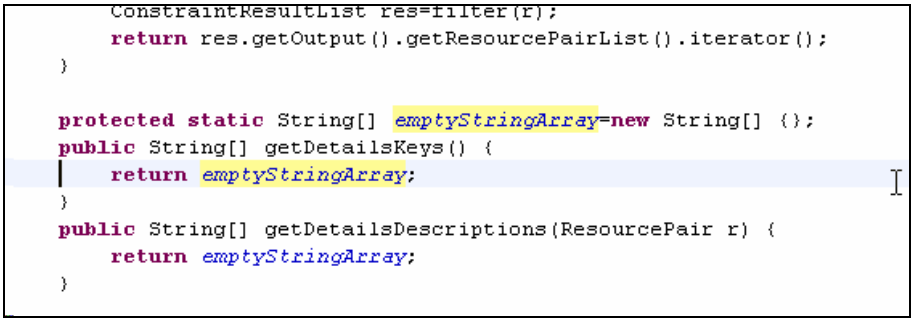
Research on actual software process [1] focuses on observing, describing, and analysing software development on an empirical level. It naturally starts bottom-up, examining in the first place micro-activities of single developers performing a subtask step-by-step. This paper presents a set of activity types and its properties which are meant to be atomic, i.e. to analyse actual programming processes there will be no need to examine even more detailed actions. The activity types are currently restricted on mainly source code changing and closely related activities (see section 3), although its concepts are general enough for describing changes on every kind of software artefact. Only single programmers are considered. No verbal communication, coordination, or discussion (which of course form an important part of software development) nor requirement elicitation, planning, or administrative tasks have been considered in the scheme.

The annotation scheme has been developed while manually analysing 2 hours and 46 minutes of programming sessions both taken from a mid-sized semantic web project [2] and very small-sized ACM programming contest training units [3].

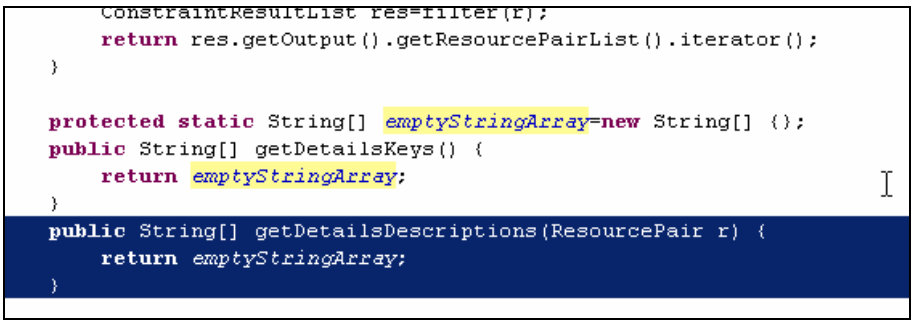
Analysing actual processes means annotating excerpts of programming sessions (available as video recordings) with operation types, e.g. programmer activities. The result is called an episode. Episodes are occurrences of operations as well as interpretations of excerpts. Figure 1 illustrates these associations using an UML class diagram notation.

Figure 1 : Episode = Interpreted Excerpt as well as occurrence of an Operation

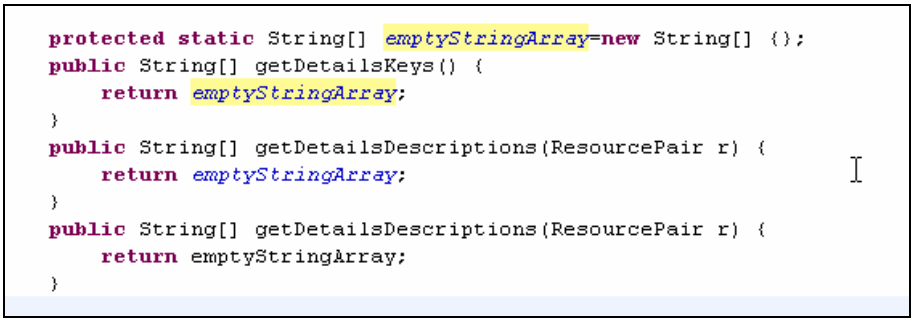
The concepts of the annotation scheme are explained using the following running example, a small code change: The programmer adds a new method called „getDetailsDescriptions()“ to class „Base“ via copying previously written code and altering it afterwards. A syntactical defect (unknown type name) has been introduced. Figure 2 shows five states during this episode. Only the covered part of class “Base” is visible.



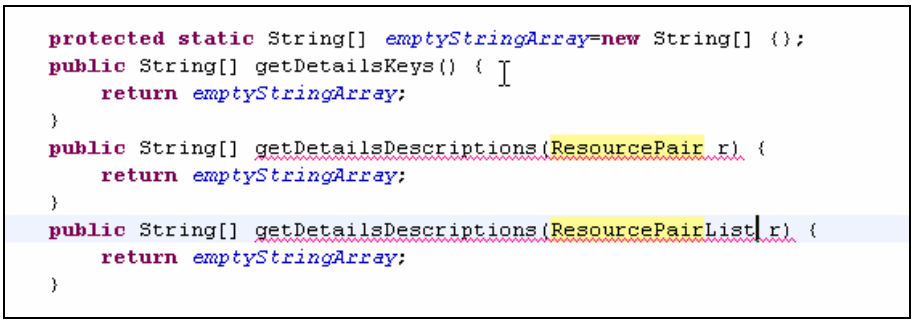
(a)



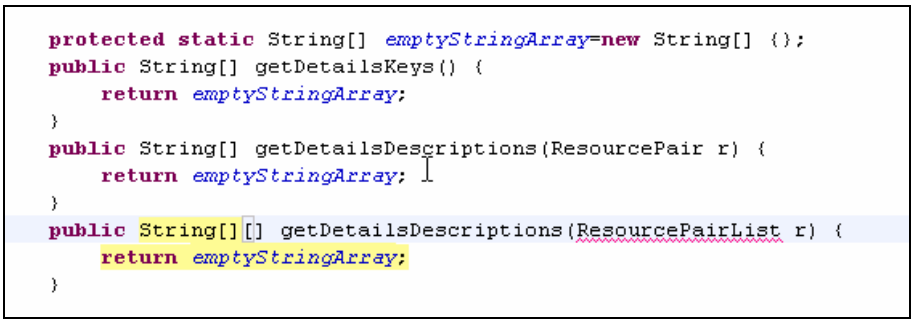
(b)



(c)



(d)



(e)

Figure 2 : Video stills from a Java code change: The programmer (a) positions cursor, (b) selects code part to be copied, (c) pastes code below the original, (d) changes parameter type, (e) changes return type. The editor (Eclipse) continuously highlights line of cursor (light grey), string matches of expression at position of cursor (yellow), and warnings/errors (curly underlining)

Figure 3 shows the complete UML class model of the representation of episodes in the annotation framework. Attributes are omitted. The following sections, which introduce the classes, will regularly refer to this model.

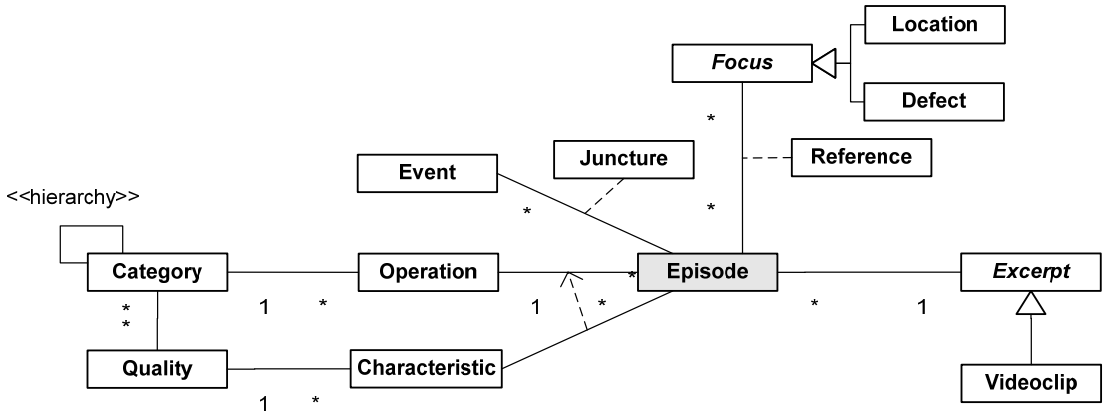


Figure 3 : General annotation model

The development of annotation (or coding) schemes is common practice in qualitative research to analyse documents, transcripts, interviews, or videos [4][5]. Only few research papers have been published on programming, see for example [6] on code comprehension. Detailed studies of programming sessions have been undertaken in research on "Psychology of Programming" [7][8][9] as well, but none fits the level of detail realized in the framework presented here.

All annotation concepts will be explained in more detail in sections 2 to 6. The paper concludes with an outlook on applications and planned work.

2. Excerpts

The annotation scheme assumes that any programming session can be split into small chunks of interesting and interpretable excerpts. The name "excerpt" is actually motivated by the analysis of video material (screen capture plus sound plus face video) of programmers at work. An excerpt is a time frame in which exactly one operation happens. (Operations are introduced in the next section.) In other words, it is the longest possible sequence without any relevant change of situation or status, i.e. no influencing event (like interruptions), no mental state change (like changing interest), and no second action (like switching from writing to reading) happens during an excerpt. Excerpts can consist of video clips, session transcripts, or a set of basic, technical events recorded automatically [10]. Excerpts have properties of time, duration, and programmer name.

The 2:46 hours of videos analysed by now have an average excerpt duration of 27 seconds ranging from two seconds to exceptional two minutes. The running example (Figure 1) is 16 seconds long. The median length is 15 seconds, because only 16 excerpts last for more than a minute. Theoretically, excerpts may overlap each other although this is not the case for the excerpts annotated so far.

3. Operations and Categories of Operations

Operations are interpretations of excerpts, i.e. they simply denote "what is going on". Operations (more precisely: operation types) are mostly activities of the programmer, including non-actions or mental actions like pausing or thinking. Throughout this paper all operations are activities of this kind. For extensibility, other operations may be introduced as well, for example programming phases summarizing tenths of programming activities.

Categories are groups of operations which form a hierarchy. Activities define the highest level category of operations. Sub-categories are core activities, inner activities and batch activities.

Core Activities

Core activities are further classified in more detailed categories. The most important kind of core activity is the *code change*. The annotation scheme identifies the following different ways of code change[1]:

- *Advancement*: In an ideal world, a programmer always makes progress in the sense that she adds functionality without the need to ever re-examine or extending what she did. In this case, all activities would be "Advancements", which is adding code as planned and making linear progress towards the next sub-goal. Naturally, starting with a new program or a new module is always Advancement. This does not mean that the code is correct or that it will never be altered or deleted. The running example episode is of the Advancement kind: new code is added to add functionality.
- *Betterment*: A "Betterment" is an unintentional change of previously written code which initially was assumed to be correct, complete, and appropriate. In most cases, simply a defect is removed. It is not a complete redesign of code due to new functional or non-functional requirements (which would be Displacement). Instead, the programmer could have done it correct in the first place, i.e. Betterments are avoidable in principle.
- *Complement*: In comparison to Advancements which explore new territory, and in comparison to Betterments which only restore initial intentions, "Complements" are planned code changes which generalize previously functional code parts, for example: The change of a String attribute type to a list of Strings because new functionality requires a more general data structure. Someone who builds up code evolutionary (starting with a simple, running version and extending it iteration per iteration) is likely to perform many Complements.
- *Displacement*: A "Displacement" is some removal of code which is not useful or necessary anymore because of a new design, dead code, change of mind, new insights, better plan, or obsolete prototypical code.
- *Embellishment*: Beautifying or refactoring code without changing its semantics is called "Embellishment". This is often done occasionally, and sometimes in parallel to some other activities like thinking or reading code.

Code documentation would be a sixth type of code change. Although not investigated by now, these code changing activities may as well be used for design documents or the like. It may have become apparent that the code changes differentiated here are not based on programming language constructs (e.g. no "new parameter introduced" or "variable name changed") but are related to programming progress and how the actual (mostly unintentional) planning process of writing/altering the document looks like [8].

It is an open issue to describe these activities more formally based on text/code manipulations.

Besides changing code, other core activity categories – which will not be explained in detail for lack of space – are:

- *Browsing*, in code or in code documentation, for a class, an operator, a method, etc.
- *Reading*: Examining requirements; Reorientation in code (e.g. after interruption); Reviewing code just written
- *Thinking* about the next step; Thinking about an event (see section 6) which just happened; Thinking about current programming problem[2].
- *Pausing* work because of other interests, or waiting for s/o or s/th. Additionally, all "doing nothing visible" activities without any clear explanation are annotated as Pausing.

---

[1] In each category the names of the operations were chosen to be equal in style but different in the first letter to ease manual annotation of coding sessions. This has resulted in some unusual but catchy terms.

[2] Of course, it is not easy to tell one from the other.

- Other kind of *Work*, like talking to others, generally using the programming environment or operating system, using the program during test, delivering code to version control, preparing test data, etc. These activities definitely require more elaboration which hasn't been possible by now because of a lack of occurrences in the available videos.

The set of core activities (like any other category of operations) may and will be expanded.

Among the core activities, mostly Advancements have been observed. This is clearly a consequence of the fact that projects at the start of development or small programming tasks have been examined. 48% of the core activities are code changes (56% Adv., 30% Bet., 10% Dis., 4% Com., 0% Emb.), thinking 21%, reading 12%, pausing 7%, and browsing 4%. The surprisingly low amount of browsing may be due to the good knowledge of the programmers and the small size of the projects. Note that only the mere occurrences are counted, not their duration. Pausing and browsing for example have a comparatively long duration and therefore allocate a larger amount of time.

Inner Activities

In many cases, a programmer does not continuously perform core activities. Often other small inner actions are done along the way during a main activity. These range from quick corrections of typos (called Re-Spell here) to typing isolated code fragments while unconsciously thinking hard about a problem (Plan-Aloud). So far, the following inner activities have been defined:

- *Tag-Along* (the term is taken from [9]) is a quick change of code just written, for example to alter the name of a recently introduced variable, although the programmer is already concerned with the next sub-goal. It can also be any other quick and short code maintenance step.
- *Work-Over* is a quick Betterment, for example changing the termination conditional of a loop while thinking about the loop's body.
- *Dust-Off* is a small Displacement (just like Work-Over is a small Betterment).
- *Look-up* is Reading and/or Browsing the code or documentation at a different place than the focus of the current main core activity.
- *Spruce-Up* is a quick Embellishment, for example correcting indentation.
- *Re-Spell* and *Plan-Aloud* as described above

Since these inner activities mostly take only few seconds of an excerpt, they are annotated in addition to a core activity. This means that the core activity (e.g. Advancement) includes one or more inner activities (e.g. Re-Spell) without precisely specifying the point(s) of time. As a result, an excerpt may be associated with more than one operation, leading to different episodes. Theoretically, it would be possible to annotate all single inner activities for new (small) overlapping excerpts, but this would be unpractical in case of manual annotation.

Batch Activities

Batch Activities are initiated but not executed by the programmer. It is *Compilation* of the code and *Building* the code as well as *Running* or *Debugging* the program, i.e. testing it. Debugging simply means to run the program in debug mode. The full debugging activity has not been investigated in more depth so far although it is an interesting and important one, including comprehension and defect removal. In fact, defect removal is annotated just like any other code change, although it probably will contain much more Betterments than Advancements.

4. Qualities and Characteristics

Operations have optional qualities. For example, code changes can be performed in different speeds. The qualities' values are called characteristics, i.e. speed may have the characteristics "fast" and "slow". Figure 3 shows that qualities are bound to categories, i.e. each operation in a category may have characteristics of the category's qualities. Qualities are inherited along the category hierarchy.

Qualities and their characteristics allow to specify an operation more precisely. Here are the ones which have been most frequently used by now:

- For code changes, the *source* denotes where the written code originally came from. The usual case is "*brain*", but *copy* (-paste) from other code locations or documentation as well as *type writing* are possible as well.
- For core activities, the *speed* can be qualified as mentioned. The characteristics are *faster* and *slower*, i.e. it is related to former speed and not interpreted in absolute quantities because it is important to detect behavioural changes.
- For code changes, the *finish* quality expresses whether the code has been left incomplete at the end of the code change – for example "if (count =", maybe caused by an interruption – or open. Open code locations are syntactically well formed but obviously not finished, like empty brackets in "if (count == 0) { }". Often locations are left open when the train of thought found a more important or task to be completed before.
- For all activities, the *pressure* (for example time or success pressure) on the programmer can be characterized as *higher*. This quality has been included because the annotation scheme has mainly been introduced to investigate coding episodes in which defects have been injected. See section 8 for more on applications of the coding annotation scheme.

5. Foci and References

Operations are not only performed *by* the programmer but also performed *on* something, i.e. the object or *focus* of operation. The most important focus is the *code* (or document) *location*. A code location is rarely an entire code file but mostly finer grained. In fact, the term is not defined in any way. What is actually called a code location depends on the level of investigation. So far, locations of class member granularity (methods, fields) have been used.

An episode may refer to more than one code location focus. A typical example is that of an Advancement activity creating a method (focus 1) and therefore extending a class (focus 2). *Creates* and *extends* are two types of *references* (Figure 3: Reference is an association class between Episode and Focus) for code locations, the others being *changes*, *discards*, and *copies*. The latter has been introduced to allow a more precise specification of code changes with Quality source = copy.

Foci are used to group operations of similar interest. For example, by using code locations coding activities can be selected for each location separately. Figure 4 visualizes about three minutes of a coding session consisting of nine episodes[3]. The blocks and their colour represent the kind of core activity, namely Advancement as green, Betterment red, Browsing yellow, and Thinking in white.

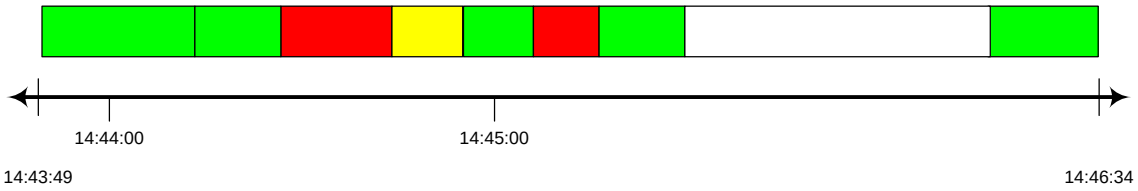


Figure 4 : Timeline visualization of some episodes

Using the references to the annotated code locations, the possibilities of exploratory analysis of the data are raised. Figure 5 separates the locations L1 to L4. Now it is possible to see that for example L1 has subsequently been corrected after advancing to other locations. This probably forms a typical behavioural pattern: Initially coding a new method and later on altering the implementation to the needs of other methods which call the initial one.

The concept of a focus is used in a general way as a kind of recurring theme during programming. Besides locations, *defects* are also annotated as foci[4]. The typical references on defects are *introduces*, *detects*, and *resolves*. Defects are not code locations (although they often can be clearly located) but simple identities which allow for analysis of a defect's life cycle. It results in episodes like

[3] Taken from the same video as the running example. Actually, block no. 9 furthest to the right *is* the example.

[4] This again was motivated by the research on causes for programming errors [11].

"*Complement in slower speed which changes method firstPass(), changes class CLIParser, and introduces defect #3*".

Other kinds of foci can be introduced as well. One such candidate is the current intention or sub-goal of the programmer in terms of the problem to be solved.

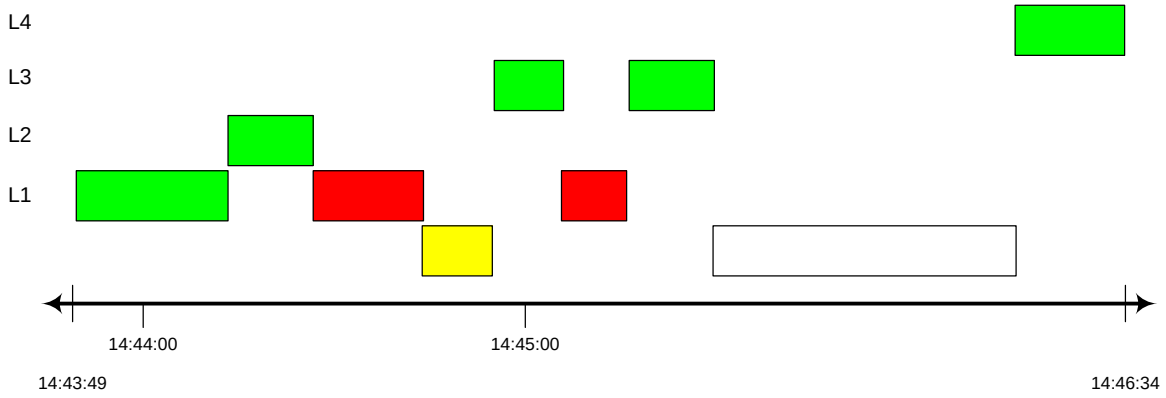


Figure 5 : Timeline visualization of some episodes grouped by code location

6. Events

Events represent some sort of impact on the programmer's behaviour. It occurs at a point of time, the *juncture*: at the *beginning* or at the *end* of an episode. The impact results at least in a change of operation. It is not a mental event but an event from outside, probably as an effect of the programmer's activity. Interesting events are:

- *Interruption*: The programmer has been interrupted from work and is forced to switch context. Interruptions have an important impact on the performance of programming [13].
- *Result of compilation*: The compiler presents a list of warnings and errors which usually catches the programmer's interest immediately. This is even the case for simple warnings by modern syntax-driven editors. A compilation result can be *error*, *warning*, or *no complaint*, which actually results in three different events.
- *Result of test*: After the program has been tested, its results can be *failure* or *no failure*.

7. Episodes and Schema Extensions

All concepts mentioned in sections 2 to 6 are associated with an Episode, which is the core kind of annotation concept. Annotating programming sessions simply means creating Episodes which contain an Operation, an Excerpt, probably some Characteristics, and optional Foci, triggered by or resulting in an Event. Capturing actual programming processes therefore means "to recognize episodes".

The example introduced in Figure 2 is a single episode. The object diagram of this example based on the general model (Figure 3) is shown in Figure 6. In a semi-formal notation it is: "*Advancement {source = copy} extends class Base, creates method getDetailsDescriptions2(), copies method getDetailsDescriptions(), introduces aDefect*" [5]

Instances based on the generic model can be divided into two levels of abstraction:

- *Schema*: The schema part of the annotation objects contains useful concepts for describing every programming session. It forms the *vocabulary*. It consists of the operations (including categories, qualities, and characteristics), events (including junctures), reference types, and the associations to each other. The set of schema objects can be extended as soon as new insights are gained as to which operations are important to analyse.

[5] The newly created method has later been renamed to getDetailsDescription**2**(), i.e. the "2" has been added. To remain the identity of this code location, it is consistently named even in pre-renaming episodes.

- *Actual process*: The actual process part of the objects contains concepts of an observed programming session. They form the *sentences*, based on the vocabulary. It consists of the episode, focus, and excerpt objects, as well as the associations to each other and to the schema objects.

The model itself (i.e. the set of classes in Figure 3) contains general, constant concepts about working/writing episodes in general. It therefore serves as a *grammar* to the annotation scheme.

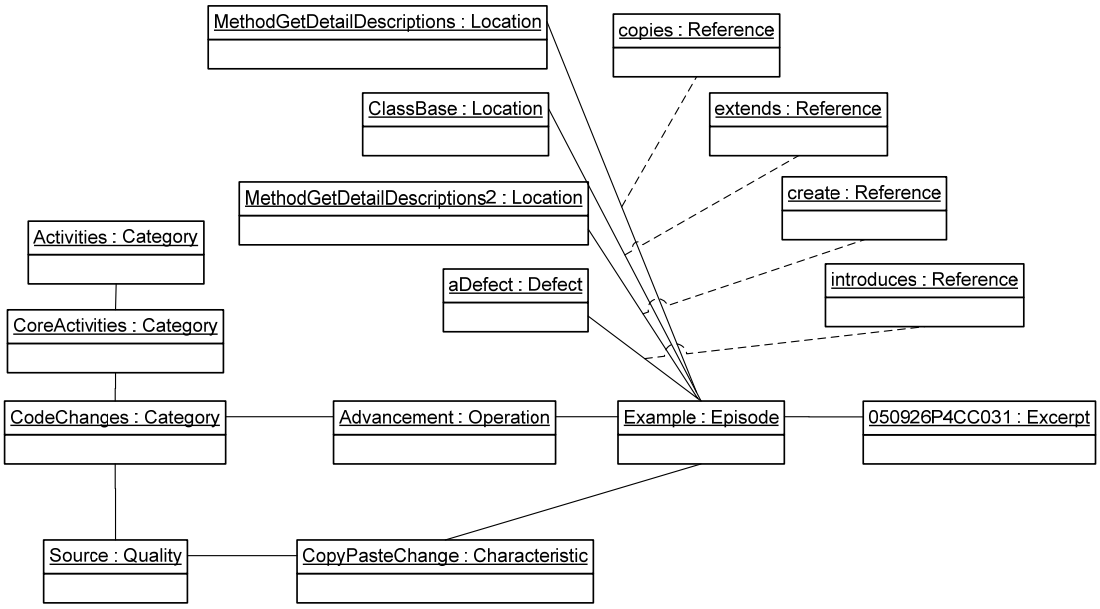


Figure 6 : Object diagram for example episode

8. Conclusions, Applications, and further Work

An annotation (or coding) scheme for programming sessions has been presented. Although it is only grounded on less than three hours of video data, more than 300 excerpts of it have been analysed and interpreted. The annotation scheme evolved during its usage and became quite stable at the end, yet the schema part will likely to be extended in the future.

Of course, it is not easy to capture actual processes. By now, it is a manual procedure, and a costly one, too: It takes 5 to 40 times the duration of an excerpt to extract and annotate a video excerpt. We attempt to automate this as far as possible [10], but some operations (like thinking), qualities (like source), or foci (like defects) are not likely to be ever automatically recognized. Without doubt, some are difficult to judge even manually. After all, it is a kind of process re-engineering which requires much knowledge about programming.

The scheme has been used to analyse defect injection scenes, i.e. the parts of programming sessions which result in defective code. The aim is to discover typical patterns of coding behaviour, circumstances, and indicators to describe those “dangerous” phases of programming [11], for example Copy-Paste-Change [14] or Trial-and-Error episodes. Only nine non-trivial defect introductions have been detected so far which is not enough to draw conclusions by now.

As to the application on discovering defect injection patterns some first hypotheses will be investigated statistically. For example, on first sight it seems that Betterments are done (as well as defects introduced) during periods of “chaotic programming”: when changing code locations often and in changing order. It will be necessary to define relevant metrics based on the sequence of episodes.

Moreover, typical patterns of consecutive episodes may provide a higher level of abstraction and annotation scheme. It may be possible to automate the detection, but more likely will provide some form of human investigation of episode visualisations like outlined in section 5. Developing a useful way to visualise the annotations will be crucial to serious qualitative analysis. Another effort will be to semi-automate the annotation process itself. To obtain a critical amount of data for analysis, however, most effort needs to be put in the actual annotation of more realistic coding sessions.

References

[1] L. Prechelt, S. Jekutsch, P. Johnson: *Actual Process: A Research Program.* Technical Report B-06-02, Inst. F. Informatik, Freie Universität Berlin, March 2006

[2] Project "Reisewissen" („Travelling Knowledge") : http://reisewissen.ag-nbi.de/en (03/02/2006)

[3] ACM programming contest homepage: http://icpc.baylor.edu/icpc/ (03/02/2006)

[4] T. Lethbridge, S.E. Sim, J. Singer: *Software Anthropology: Performing Field Studies in Software Companies*, http://citeseer.ist.psu.edu/263534.html (03/02/2006)

[5] Carolyn B. Seaman: *Qualitative Methods in Empirical Studies of Software Engineering.* IEEE Trans. on Softw. Eng., 25 (4), July/August 1999, pp. 557-572

[6] A. von Mayrhauser, S. Lang: *A Coding Scheme to Support Systematic Analysis of Software Comprehension.* IEEE Trans. on Softw. Eng. 25 (4), July/August 1999, pp. 526-540

[7] Jean-Michel Hoc et.al (ed.): *Psychology of programming.* Academic Press 1990

[8] Simon P. Davis: *Models and theories of programming strategy.* Int. Journal Man-Machine Studies (1993) 39, pp. 237-267

[9] W. D. Gray, J. R. Anderson: *Change-Episodes in Coding: When and how do Programmers change their Code?* Empirical studies of programmers: Second workshop 1987, pp. 185-197

[10] ECG homepage: http://projects.mi.fu-berlin.de/w/bin/view/SE/ElectroCodeoGram (03/02/2006)

[11] Research project on "Micro-process analysis for avoiding programming errors" homepage: http://projects.mi.fu-berlin.de/w/bin/view/SE/ErrorHome (03/02/2006)

[12] Hackystat homepage: http://csdl.ics.hawaii.edu/Tools/Hackystat/ (03/02/2006)

[13] I. Burmistrov, A. Leonova: *Do interrupted users work faster or slower? The micro-analysis of computerized text editing task.* In: J. Jacko and C. Stephanidis (Eds.) Human-Computer Interaction: Theory and Practice (Part I) – Proceedings of HCI International 2003, Vol. 1. Mahwah: Lawrence Erlbaum Associates, pp. 621-625

[14] M. Kim, L. Bergman, T. Lau, D. Notkin. *An Ethnographic Study of Copy and Paste Programming Practices in OOPL*, Int. Symp. on Empirical Software Engineering, August 2004, pp. 83-92