ElectroCodeoGram: An Environment for Studying Programming

Freie Universität Berlin, Institut f. Informatik, Takustr. 9, 14195 Berlin, Germany
frank@schlesinger.com
jekutsch@inf.fu-berlin.de

Frank Schlesinger
Sebastian Jekutsch

Abstract Research on programming behaviour requires the analysis of a huge amount of observational data. It would be helpful to have tool support in collecting and summarizing these data containing basic programming events. This paper introduces ElectroCodeoGram (ECG), a modular framework consisting of event collectors (called sensors), pattern recognizers (called modules), and other infrastructure to offer help in this respect.

1 Introduction

Actual process research In software engineering the term "process" is usually used in a prescriptive manner. It defines the order and activities of software development. Developers are encouraged to adopt their work to process model guidelines to improve productivity or product quality. The term "process" can, however, also be used in a descriptive manner: it then describes what software developers are actually doing, rather than telling what they should do. We call this the "actual process" [1].

The research projects in our workgroup focus on actual processes, especially "micro-processes" [2]: Studies on developer documentation, pair programming, and programming errors. They all have in common that they rely on empirically gained data. A micro-process describes the sequence of programming activities on a very fine-grained level. The activities studied in micro-process research are of very short duration: Opening of a file, running of the program, renaming of a method, copying and pasting of code blocks, etc.

Studying programming behaviour The goal of micro-process research is to discover patterns in the sequence of events which describe typical programming behaviour. Those behavioural patterns are called "episodes" [6]. For the project on programming errors, the basic assumption is that some programming behaviour leads to code of better quality and with fewer defects than others.

For instance, if you need similar functionality in two different locations of your code, you have at least two alternatives: (a) You could copy a code fragment that provides the desired functionality from the first location and paste it into the second one. You would normally need to make some minor modifications to the pasted block afterwards. This kind of approach corresponds to what we call a "copy-paste-change episode". (b) Alternatively, you could also go the refactoring-way and define a new method providing the desired functionality, which is then invoked from both locations with different parameters. This is what we refer to as a "refactoring episode". Most programmers would tend to say that the second alternative is often the better one. In the first case, you could forget to make all necessary modifications to the pasted code block in a consistent manner. In addition, later changes of requirements may lead to a change in the original code-block. Will the programmer remember to check all pasted blocks to see if they need modifications?

Still, the question remains open whether copy-paste-change episodes really result in more code defects than refactoring episodes. We need to collect data from many different programming sessions, extract specific episodes and defect insertions, and finally calculate correlations between them to tell "good" from "bad" behaviour in terms of the probability of producing defects.

Need for tool support The idea of micro-process recording and episode analysis is simple: Record the micro-process of a programmer as a sequence of events and recognize relevant episodes of programming behaviour. The actual task, however, is facing at least two non-trivial problems. First, micro-process data (the basic programming events) should be recorded and analyzed mostly automatically. The pure amount of data for even only an hour of coding would be too large to handle manually.

The second problem is that often the interesting entities of the micro-process research are not well known at the beginning. Which activities of the programmer are relevant to the current study? Which episodes must be discovered in the stream of events? Actual process research requires an exploratory approach. The data will be analysed and consolidated repeatedly in many different ways and from different points of views. This again calls for some tools to aid in studying programming.

2 ElectroCodeoGram = Lab + Sensors

ElectroCodeoGram (ECG) [3][4] is a software system, which is intended to support the actual process and micro-process research; it helps researchers in solving both difficulties mentioned above. Its main purpose is to record micro-process data and to support analysis and episode recognition in the micro-process event stream.

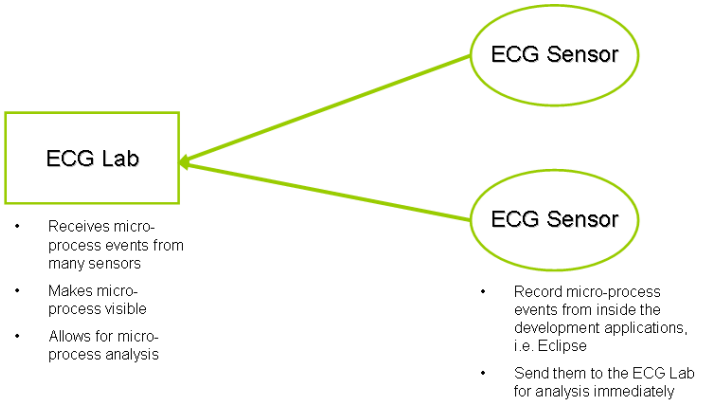

Figure 1: The main components of ECG: The ECG Lab and ECG sensors

ECG is a distributed system made up of sensors (recording the events) and a centralized data collection and analysis system. Sensors are integrated into the tools the programmer uses. For instance, we developed an ECG sensor for Eclipse (www.eclipse.org), which is implemented as an Eclipse plug-in. These sensors collect relevant events of globally defined event types from the interaction of the programmer with the IDE. Every recorded event forms a "micro-activity" and is sent to the "ECG Lab" (Figure 1).

The ECG Lab is a server, which concurrently receives micro-process event data from multiple sensors. Thereby, the ECG Lab can reside on a different computer and the event data is transported over network sockets or SOAP. Of course, both the sensors and the ECG Lab can be run on the same computer.

The Lab can also be used for automated event analysis. While a programmer is coding, the stream of micro-process events can be analysed at the same time. If the Lab runs on the programmer's computer, this feature offers immediate feedback to the programmer informing about suspect programming behaviour. For example,

when the programmer copies and pastes one and the same code fragment more than three times, a warning could be issued.

The user interface of the Lab gives basic access to the event-stream. It is possible to observe which events are recorded. The main functionality is hidden in different modules (Figure 2).

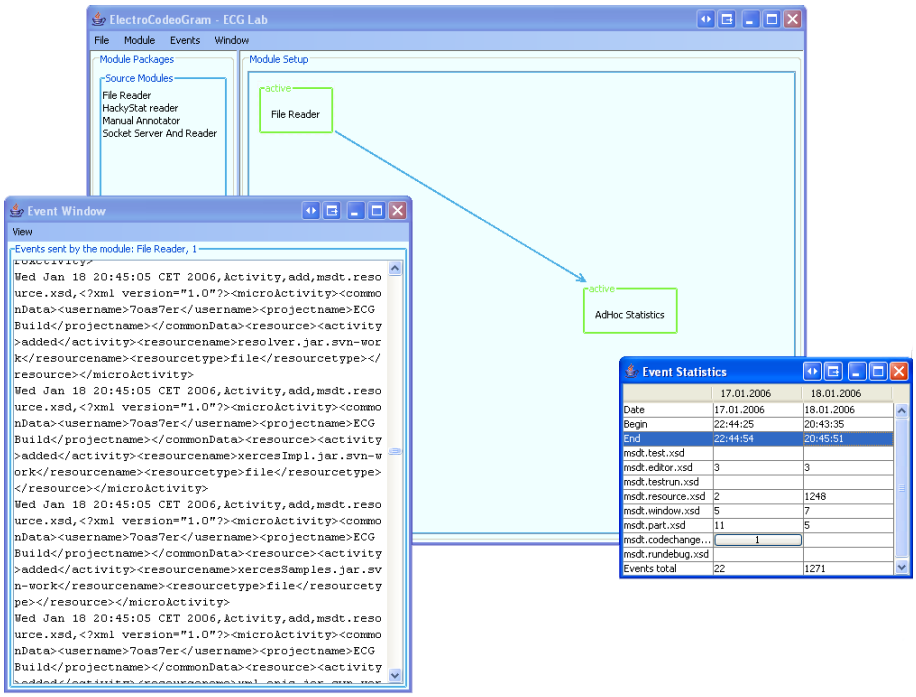

Figure 2: The ECG Lab GUI with an event window and the output of the statistics module

3 ECG modules

The ECG Lab contains a set of “ECG modules”. Each module takes an event-stream as its input, provides a special kind of event processing functionality depending on its implementation, and finally produces an event-stream as its output. Currently, the following modules are available:

- episode recognizers, which detect file edit episodes and window activities

- a module to present statistics on a recorded event, like event count per type and day
- a module to filter out certain classes of events from the stream
- various modules to write out the event stream to disk or other locations and read them back in again
- a module, which receives the events from the sensors

The ECG modules can be linked interactively to each other in the Lab's GUI, resulting in a kind of module graph in which the output of one module is the input of another module (Figure 3). The micro-process events enter the ECG Lab using "source modules", which receive them from outside the ECG Lab, the sensors or a file. Reading in events from a file allows to analyse micro-processes repeatedly using different Lab configurations, i.e. filters, recognizers, writers, etc.

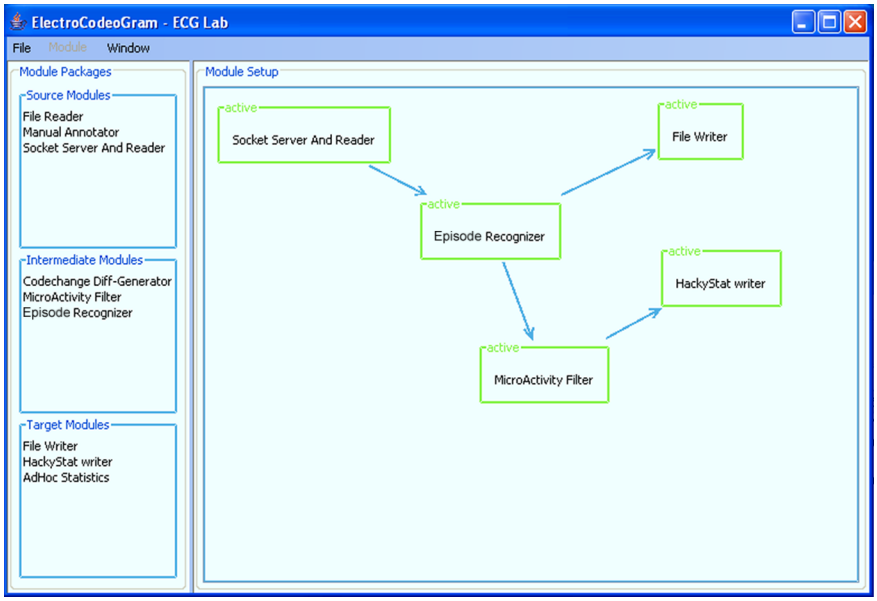

Figure 3: An example module graph in the ECG Lab

In case an "intermediate module" analyses the event-stream, its results are added to the event-stream afterwards. Consider a module which recognizes copy-paste-change episodes: The module receives the events sequentially, waiting for a specific pattern of events. After the occurrence of the pattern, the module creates a

new event, which now contains the information that a copy-paste-change episode has been recognized (Figure 4).

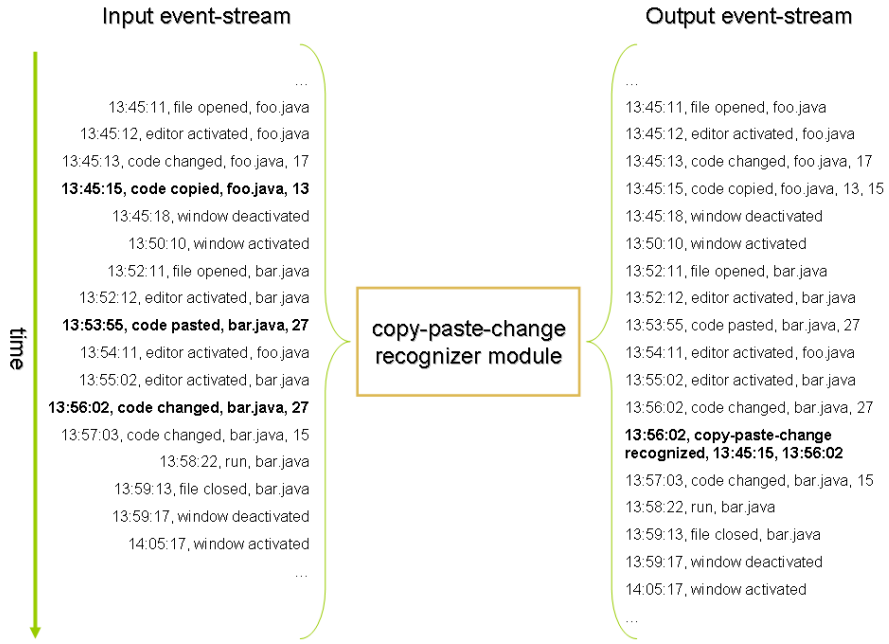

Figure 4: The functionality of an episode-recognizer module: The input event-stream is observed for an episode-pattern. If one is found a new event is created and inserted into the event-stream.

The intermediate modules can be connected to each other in a pipeline manner. Therefore an analysis may depend on the results of a previous analysis. Finally, events can be passed on to "target modules", where the event-stream is written out sequentially to a location outside the ECG Lab, such as into a text file. It is the target module which would be responsible to give feedback alerts to the programmer, as has been explained in section 2.

Since it is feasible to connect multiple modules to another module, you can have many analyses running in parallel. By connecting them to different target modules, the different analysis results can be stored into distinct locations.

ECG provides a module programming interface that enables the implementation of analysis modules independent of other modules. At its physical level, a module is a folder in the ECG Lab's module directory containing Java classes along with a

properties file. When the ECG Lab is started, the modules are loaded and are ready for use.

4 Micro-process event types

Internally, the micro-process events are represented in XML, based on XML schemata. Schemata are used to validate the event data received from the sensors. Each micro-process event consists at least of a type, a developer name, and a time stamp. Different event types require different additional data, like file names, or changed code. The basic event types are: Window de-/activation, editor de-/activation, run/debug, and file change (including the changed contents). Additional event types specifically reports Eclipse usage. Furthermore, modules may define new event types, for example episode-like events such as copy-paste-change events, which are not generated by a sensor but by intermediate analysis modules.

Here is an example of how ECG is currently being used in a research project on programming strategies in pair programming. Eclipse usage data were captured for six pairs of programmers and three single programmers, resulting in nine programming sessions of four hours' duration each. All programmers solved (or at least tried to solve) the same programming problem using a similar environment. The collected micro-process data contains the usual basic events, including code changes, run/debug triggers, and window/file activities.

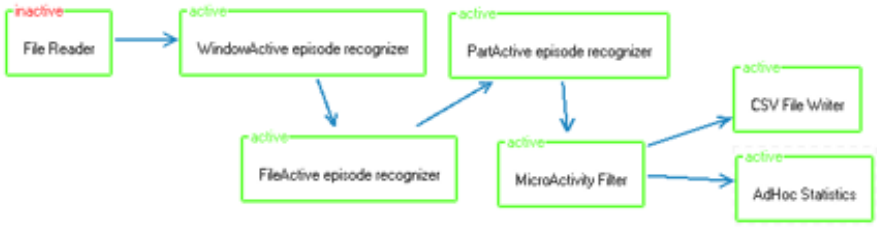

Figure 5: A module-setup in the ECG Lab used to investigate programming strategies in pair programming

Three episode recognizers were created to compile new simple episodes from the raw data: file activity (including a flag marking whether the file has been changed or not), part activity (part is a sub-window of Eclipse), and window activity (which indicates whether Eclipse was active or not). The raw events (read in by using the file reader module) were filtered because they were of no interest to the analysis

with the exception of the run/debug event. The resulting events were written in a newly introduced target module which writes CSV-like files. The statistics module is used as a progress meter (Figure 5).

The episode recognizers defined new event types not known by the sensors. The resulting file contained only these new events plus the run/debug events. The files were read by statistics software as well as Excel to provide some overview on what happened during the programming sessions, and to visualize how the successful subjects differed from the less successful programmers in their file changing and compilation/testing behaviour.

5 Enhancing ECG

At the moment, the analysis functionality of ECG is quite limited, and most of the modules are prototype implementations. Some of our current thesis projects aim at extending the ECG with new modules and sensors:

- The copy-paste-change episode recognizer is still under development. It requires new event types and episode recognizers as presented in section 3. Its future application will warn the programmer when an original code-block has been changed but the copies have not.
- Based on the assumption that an interruption of the programmer leads to more defects in the code, the question arises of how it is possible to automatically recognize when the programmer has been interrupted from his work. One project concentrates on the development of a sensor which observes the active window in the programmer's computer desktop to report active windows which are configured to be non-programming related. Additionally, a module shall automatically recognize interruption episodes in the micro-process of a programmer.
- Another episode recognizer under development aims at detecting "trial-and-error episodes", i.e. programming phases in which a code location is changed several times and after each change the program is given another run.
- New source and target modules for the ECG are being developed in a project to make the ECG interoperable with relational database systems. If the micro-process events are mapped and written into a relational database, highly interesting queries on the micro-process are possible: Find the longest "trial-and-error episode" in the last week for programmer Frank, for instance.

- Efforts are put also into the visualization of the stream of events and episodes. One possibility is to depict them in a timeline-like graph of activities at various code places, the other to provide a kind of movie of consecutive code changes at a single code location to explore code changing activities.
- On the other hand, we need to integrate videos into the ECG event stream because of course not all important events and episodes can be recognized automatically. While playing synchronized screen captures from the development environment as well as videos of the programmer herself, one can add these events and episodes manually into the event stream using an intermediate module which is already available. This will also help in evaluating the episode recognizers, i.e. it will be possible to compare the result of the recognizers with the video.

6 Related work and summary

Besides ECG, there are other software systems which automatically collect and analyse coding events. One of them is Hackystat [5], a system intended to collect PSP-like data from various sources. About a dozen sensors collect events for Hackystat from many development environments, editors, and even office programs. All of these send events to the Hackystat server, a web-application analysing the events and generating web-sides with tables and charts of the event data.

EletroCodeoGram was build with respect to Hackystat version 6 standards. It is interoperable with Hackystat, i.e. every Hackystat sensor can be used to collect data for the ECG, and the ECG can send micro-process data and analysis-result events to a Hackystat server for storage. Plans are to upgrade to Hackystat version 7.

Hackystat has lately been extended with "Zorro" [7], which supports the recognition of behavioural patterns in recorded events. The main difference to ECG is that Zorro fully relies on the Hackystat event classes, which are far less fine-grained than the micro-process events recorded by ECG. Thus, Zorro presents the programming behaviour on a much higher level. Neither Hackystat nor Zorro offer "on-the-spot" feedback to the programmer and both are more heavy-weighted in terms of installation requirements.

ECG is a new system to record and analyse programming behaviour. It is currently in use in various software engineering experiments and is extended by multiple projects in our workgroup. It comes with a modular analysis framework and programming interface. ECG is both a measurement tool for empirical investigation

of programming behaviour and a "box of bricks" to work with actual programming micro-processes. It is available as open source [8].

References

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

[2] Jekutsch S.: Micro-process of software development.
URL: http://projects.mi.fu-berlin.de/w/bin/view/SE/MicroprocessHome
(Oct. 5. 2005)

[3] ECG home page:
http://projects.mi.fu-berlin.de/w/bin/view/SE/ElectroCodeoGram

[4] Schlesinger F.: Protokollierung von Programmiertätigkeiten in der Eclipse-Umgebung. Diploma thesis, Inst. f. Informatik, Freie Universität Berlin, 2005 (in German)

[5] Johnson, Philip M.: Project Hackystat: Accelerating adoption of empirically guided software development through non-disruptive, developer-centric, in-process data collection and analysis, Technical Report csdl2-01-13, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, November 2001

[6] Jekutsch S.: An Annotation Scheme to Support Analysis of Programming Activities. Workshop on Ethnographies of Code, Univ. Lancaster, UK, March 2006

[7] Kou H., Johnson P.: Automated Recognition of Low-level Process: A Pilot Validation Study of Zorro for Test-Driven Development, Technical Report csdl2-06-02, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, May 2006

[8] ECG developer home page and repository:
http://developer.berlios.de/projects/ecg/