RefactorPad: Editing Source Code on Touchscreens

Felix Raab
University of Regensburg
Chair for Media Informatics
felix.raab@ur.de

Christian Wolff
University of Regensburg
Chair for Media Informatics
christian.wolff@ur.de

Florian Echtler
University of Regensburg
Chair for Media Informatics
florian.echtler@ur.de

ABSTRACT
Despite widespread use of touch-enabled devices, the field of software development has only slowly adopted new interaction methods for available tools. In this paper, we present our research on *RefactorPad*, a code editor for editing and restructuring source code on touchscreens. Since entering and modifying code with on-screen keyboards is time-consuming, we have developed a set of gestures that take program syntax into account and support common maintenance tasks on devices such as tablets. This work presents three main contributions: 1) a test setup that enables researchers and participants to collaboratively walk through code examples in real-time; 2) the results of a user study on editing source code with both finger and pen gestures; 3) a list of operations and some design guidelines for creators of code editors or software development environments who wish to optimize their tools for touchscreens.

Author Keywords
Editor; source code; IDE; gestures; pen; touchscreen; tablet; surface; refactoring.

ACM Classification Keywords
D.2.3 [Software Engineering]: Coding Tools and Techniques - Program Editors; D.2.6 [Software Engineering]: Programming Environments - Interactive environments.

INTRODUCTION
As devices with touchscreens have become mainstream, an increasing number of application domains have taken advantage of interaction via multi-touch and gestures. One of the applications areas that has remained comparatively cautious with respect to widespread use of new interaction paradigms is the field of software engineering: most of the existing development tools like integrated development environments (IDEs) heavily rely on keyboard and mouse interaction in the traditional GUI style and have yet to be optimized for touch-enabled devices. In comparison with other domains, development tools stand out due to *feature-rich user interfaces* or, for developers reluctant to use graphical editors, reliance on *efficient text input*. Both usage patterns call for input techniques that do not hinder productivity when those tools need to be compatible with touchscreens in the future. Some tablets, for instance, provide high-quality text rendering that might work well for code reading and maintenance tasks. Entering and editing source code, however, is challenging without hardware keyboards.

So far, research in this field has concentrated on creating new development environments that radically differ from traditional desktop environments, in some cases by integrating visual programming concepts [7]. Despite the benefits of improving overall interaction, this approach might suffer from low acceptance among developers who are used to development environments as well as programming styles in which they have become proficient over the years. In addition, porting existing tools to multi-touch interaction is challenging: on the one hand, features cannot be simply carried over and applied to touchscreens. Codebases and user interface concepts would need a considerable amount of rework to be viable on such devices. On the other hand, pure text-based environments and editors require efficient keyboard input. While some advances in touch-typing research can improve certain aspects, almost all currently available devices still provide standard on-screen keyboards. Entering and editing large amounts of text for programming tasks can quickly get difficult and time-consuming without hardware keyboards.

Rather than fundamentally change development tools by introducing workbenches with new user interface concepts, we attempt to enhance standard text-based editors with gestural interaction. New code has to be entered via the on-screen keyboard as usual. However, code selection, editing and refactoring is supported through gestures which take programming language syntax into account. Such gestures take the place of hotkeys in traditional interfaces and therefore make common code editing tasks easier to perform on touchscreens. Since code is generally read more than it is written [8], maintenance-oriented development tools might be well-suited for portable, touch-enabled devices. Furthermore, maintenance activities such as refactoring have been shown to play a significant role in the development process. For instance, up to 70% of the structural changes of the Eclipse IDE source code can be attributed to refactoring [17]. It has been reported that

EICS'13, June 24–27, 2013, London, United Kingdom.

Microsoft uses about 20% of their development efforts for code rewriting [12].

Prior to implementing a working prototype of *RefactorPad,* we have conducted a user study to determine which gestures programmers find convenient for common maintenance tasks in a code editor. In addition, we were interested in pen or finger input preferences and in what the respective performance characteristics were. For this purpose, we have created an interactive and collaborative test setup that allowed us to walk through code examples with participants in real-time. The results of the study can be used as guidelines for implementers of touch-enabled code editors. Moreover, our test setup might be useful for other research projects that examine interaction in software development tools.

We outline related work in section 2, describe how we identified relevant editor operations in section 3 and show the test setup and experiment in section 4. In section 5, we present our results and conclude with design recommendations derived from our experiments.

RELATED WORK

In this section, we highlight some of the more recent research projects that are related to our work as they both present software development tools and integrate *natural interaction* [15] methods into their systems.

Although the project *Code Bubbles* [1] has not been built for touchscreens, it has introduced novel user interface concepts for understanding and maintaining code. The system abandons the file-oriented nature of existing tools and instead shows connected source code fragments as bubbles on a canvas. Editable fragments are grouped into simultaneously visible working sets that have shown to significantly reduce the time spent navigating and the time needed to complete code understanding tasks. A similar project, *Code Canvas* [3], leverages spatial memory to reduce disorientation. Using connected documents, semantic zoom and information overlays, it serves as an interactive map for developers. Since both projects share some ideas, a collaboration finally lead to the commercial tool *Debugger Canvas* [4]. A map-like zoomable surface supports debugging by displaying call paths and execution traces in a set of connected bubbles. Developers can then step back and forth through the code and visually explore relationships. The tool is currently used as a separate mode within the main IDE window. Since the previously mentioned projects use a zoomable canvas and do not rely exclusively on traditional user interface elements, they might work well on touchscreens when support for multi-touch interaction and gestures is added.

CodePad [10] provides interactive spaces for various programming-related tasks on secondary multi-touch-enabled devices. The devices are connected to the main IDE and are meant to support development scenarios such as refactoring, visualization or navigation. While it was mainly introduced as a vision, a prototype demonstrates some of their interaction concepts. *Code Space* [2] takes the application of natural interaction even further by enabling teams to use in-air-gestures and cross-device communication at developer meetings. *Touching Factor* [5] and *TouchDevelop* [13] present solutions for writing code on small mobile screens. However, in order to enable efficient input of code, both projects limit developers either to a certain programming language or to a specific syntax, enhanced by predefined code blocks.

IDENTIFICATION OF EDITOR OPERATIONS

Since the test system is built upon a standard code editor component that is not coupled to idiosyncrasies of certain programming languages, we first compiled a list of common editor operations that participants later had to perform during the study. Even though we did not use a strictly systematic approach of identifying these operations, we are confident that our choices reasonably represent general usage since we 1) examined some of the major editors, 2) took the personal experience of the authors and colleagues into account and 3) evaluated qualitative feedback during and after the study which did not reveal any important commands that were missing. For 1) we mainly examined the "Edit" menus of *Eclipse*, *Visual Studio*, *Xcode* and the popular text editor *Sublime Text*. Table 1 shows a list of operations used in the study.

Basic Operation	Refactoring Task
Move Caret Select Identifier Select Multiple Identifiers Select Line Select Multiple Lines Select Block Move Lines Duplicate Line Delete Line Toggle Comment Copy/Paste Undo/Redo Goto Method Declaration	Extract Method (Without Locals) Extract Method (With Parameter) Inline Method Inline Temp Replace Temp With Query Introduce Explaining Variable (Extract Local) Rename (Multiple Variables)

Table 1. List of basic operations and refactoring tasks that we selected for participants to perform in the study.

The second part of the list includes common refactoring commands. In addition to the approach we used for identifying basic edit operations, we took recent research of refactoring practice [6, 9, 14] into account. As a result, this list contains some of the refactoring tasks that are regarded as frequent and important based on interviews with developers and collected usage data. The list could be extended by various other commands, however, we did not want to further increase the number of tasks.

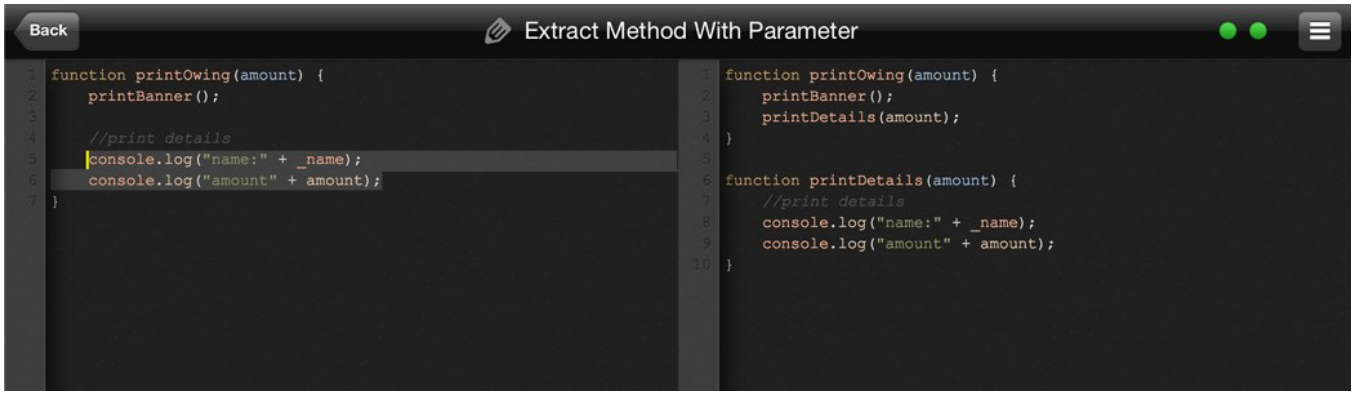

Figure 1. Editor view on the participant's tablet. The caption displays the current task, the left pane shows the initial code with highlights for emphasis and the right pane the desired result. An enhanced view with task instructions, controls to toggle the on-screen keyboard and touch/pen events in an overlay is shown on the experimenter's laptop.

TEST SETUP AND USER STUDY

Our test system consisted of two main parts: an editor running on an iPad 3 tablet showing JavaScript source code and a second, connected editor running on the laptop of the experimenter (Figure 1). We selected JavaScript, a weakly typed language, to reduce participants' mental load regarding issues such as variable declarations, return types etc. By means of a socket connection between the two systems, all touch events on the tablet and key press events of pen buttons were visualized as overlay on the experimenter's editor. In addition, the experimenter could act as a "wizard" and control different aspects on the tablet editor in real-time: modifications of the source code, selections of parts of the source code, cursor position, scrolling to certain lines, and showing or hiding the on-screen keyboard were all directly reflected on the tablet editor. A split view on both systems showed the initial state of the source code on the left side and the desired state on the right side. In order to ensure that all participants received the same instructions, additional notes were displayed on the experimenter's system for each task. This somewhat resembled a "Wizard-of-Oz" experiment except that the participants were fully aware of interacting with a remotely controlled system.

Using this setup, the experimenter could track all tablet interaction on the laptop. At the same time, we could better introduce each code example by highlighting certain code lines, thereby avoiding inconvenient pointing on the small screen in front of the participant. For later analysis, all interaction events were logged to a database on the tablet. The pen used in this study was a *Adonit Jot Touch* with two hardware buttons and a transparent touch-disk attached to the pen tip. Since not all characteristics of interaction can be reconstructed from logged touch events, we captured the area around the tablet on video so that the participant's hands and pen usage could be seen.

Participants

All participants filled in a questionnaire before the actual test. They were asked to specify their experience in certain programming languages, IDEs and their usage of devices with touchscreens. We recruited 16 participants (14 male, 2 female), aged between 21 and 32 years (Mean: 24), all right-handed. While all but one of the participants indicated (on a 5-Point Likert scale) that they use devices with touchscreens "always" or "frequently", 9 stated that they "never" use a pen for input. 12 participants had between 2 and 5 years of programming experience, 2 more than 10 years. 11 participants were "quite experienced" in the programming language Java, 4 selected "very experienced". As for JavaScript, 7 participants indicated "quite experienced" and 4 "very experienced". 10 participants were "quite experienced" in using the Eclipse IDE, 2 "very experienced". In addition, participants named programming languages and IDEs in which they were at least "somewhat experienced": PHP (8), C++ (7), C (5), C# (5), Visual Studio (5), NetBeans (4) and Objective-C (3).

Procedure

The procedure itself was mainly based on a "guessibility study" by Wobbrock et al. [16]. They achieved good results by showing users the effect of surface gestures and then letting them perform their cause. Since the test system did not respond to user input and accepted all input, the users' behavior was not affected by technical aspects such as gesture recognition. In our study, participants were first introduced to the test setup and could then try a demo task. Each of the 20 different tasks had to be done once with the pen and once using only normal touch interaction without the pen. Consequently, each participant completed 40 tasks in fully randomized order. The total number of tasks performed was: 16 participants x 20 tasks x 2 input types = 640 tasks. Participants took 75 minutes on average (including filling in the questionnaires).

A single task consisted of the following stages: First, the experimenter introduced the code example using the previously mentioned features of the test setup and made sure that the participant understood both the initial state of

the source code and the desired state (instruction phase). The participant should then try to find a suitable gesture while thinking aloud (preparation phase). As soon as the participant was ready to articulate the gesture again (articulation phase), pressing the title button started the recording of this phase, and another press stopped recording and displayed two post-task questions. Similar to the study in [16], the first question asked if the participant thought the performed gesture was a "good match for its intended purpose" ("goodness" on 7-Point Likert scale). As for the second question, we used the SMEQ (Subjective Mental Effort Question) version developed in [11] where users should indicate perceived effort by moving a slider on a scale ranging from "not at all hard to do" to "tremendously hard to do". This scale has been shown to be reliable and easy for participants to use in its interactive form. After all tasks had been performed, the test persons filled out a final questionnaire indicating which input method they preferred (pen, fingers or both) and which commands they frequently use in their development environments.

RESULTS

Agreement

In order to classify performed gestures and determine agreement scores, we examined all video captures and visualized touch events. Agreement was calculated using the same formula as in [16], using the number of participants, of gesture classes and of participants in each class. Figure 2 shows an example of all combined touch events for the task "Select Multiple Lines". This figure clearly shows two prevalent gesture classes: one group selected lines by swiping over the lines numbers in the gutter on the left side of the editor, the other group swiped across the code block from top left to bottom right. For this example, the final gesture was the gutter swiping gesture since it was used by the highest number of participants and did not conflict with other interactions.

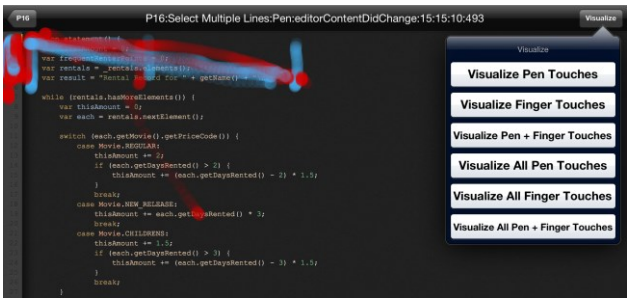

Figure 2. Visualization tool on the experimenter's system showing two patterns for the task "Select Multiple Lines".

Overall, agreement scores were lower (Mean: 0.20) than in [16] which might be due to the more complex application domain in our study. Users generally agreed most on selection gesture for identifiers, lines and blocks, "Move Caret" and "Move Lines".

Goodness – SMEQ – Agreement – Articulation Time

The relationships between the two post-task values for "goodness" and SMEQ, the calculated agreement score and the measured articulation time are illustrated in the two bubble charts in Figure 3. The diagrams show that the most agreed upon gestures were those that users perceived as good matches and least effortful. Further, those gestures were also articulated fastest. This is contrary to some of the results in [16] where articulation time did not affect goodness ratings and gestures that took longer to perform were perceived as easier. We also got different results for the number of touch events: gestures with more touch events were perceived as more effortful in our study (but did not have lower goodness ratings). Again, we suppose that these differences are due to different target groups and application domains. We could also confirm previous results: Better gestures are apparent to participants more quickly (less preparation time) and popularity (high agreement) can identify better gestures.

Figure 3. Left: Bubble chart showing aggregated values for gesture goodness (vertical), SMEQ (horizontal) and agreement (size). Right: Bubble chart showing aggregated values for goodness (vertical), SMEQ (horizontal) and articulation time (size).

We could not detect significant differences between pen and finger interaction in any of the mentioned values.

Input Preference and Frequently Used IDE Features

In the post-study questionnaire, 44% of the participants chose the pen as their preferred input method, 25% chose interaction with fingers and 31% preferred mixed pen and finger interaction. Since in the pre-study questionnaire, 56% said that they never use a pen for touch input, this somewhat suggests that support for pen interaction might be a worthwhile addition to touch-enabled code editors. IDE features that participants frequently use at their own judgement, are (*number of mentions in brackets*): Rename (6), Auto-complete (5), Navigation to method or class (5), Auto-format (4), Save (3), Extract method (2), Create new method (2).

Qualitative Observations

During the study, we observed that the users' mental models are strongly influenced by interaction concepts of mobile operating systems. Most of the participants could easily be identified as "Android users" or "iOS users". Additionally, users frequently asked for context menus since they either could not think of a suitable gesture or found a menu more convenient in certain cases. At the same time, however, they expressed their dislike for menus that contain too many items. Some participants were concerned that selection and gesture recognition might not be precise enough in a working system, leading to a lot of re-selection and adjustments in the editor. Most users seemed to prefer one-handed gestures and used multi-touch interaction only conservatively (only few gestures were performed with more than two fingers). According to participants' comments, the pen was generally perceived as more accurate than interaction with the fingers. Users often decided to perform the same gesture for both the pen and finger version of the task. The two hardware buttons of the pen were sometimes used as left and right mouse buttons.

As far as specific refactoring operations are concerned, users generally seemed to find it easier to extract than to inline code. Some inline operations resulted in sequences of unnecessary steps to complete the task. For users without prior knowledge of inline refactoring, it was not apparent that this transformation could be automated and hence only needed a gestural trigger to be initiated.

Design Recommendations

Based on results from the user study, we propose a set of gestures (Figure 4) for the operations used in this study. This could serve as starting point for implementers of touch-enabled code editors. The set also shows some user interface elements that should be considered as interactive zones: For instance, the majority of users chose the gutter with line numbers as selection target for multiple lines and code blocks by swiping over the corresponding area. Without involving users, we probably would not have predicted this area to be a popular line selection target. Although we tried to remove context menus from the set, integrating more commands would certainly need some form of touch-optimized menu or sidebar that could be displayed on demand. As another subtle, yet important,

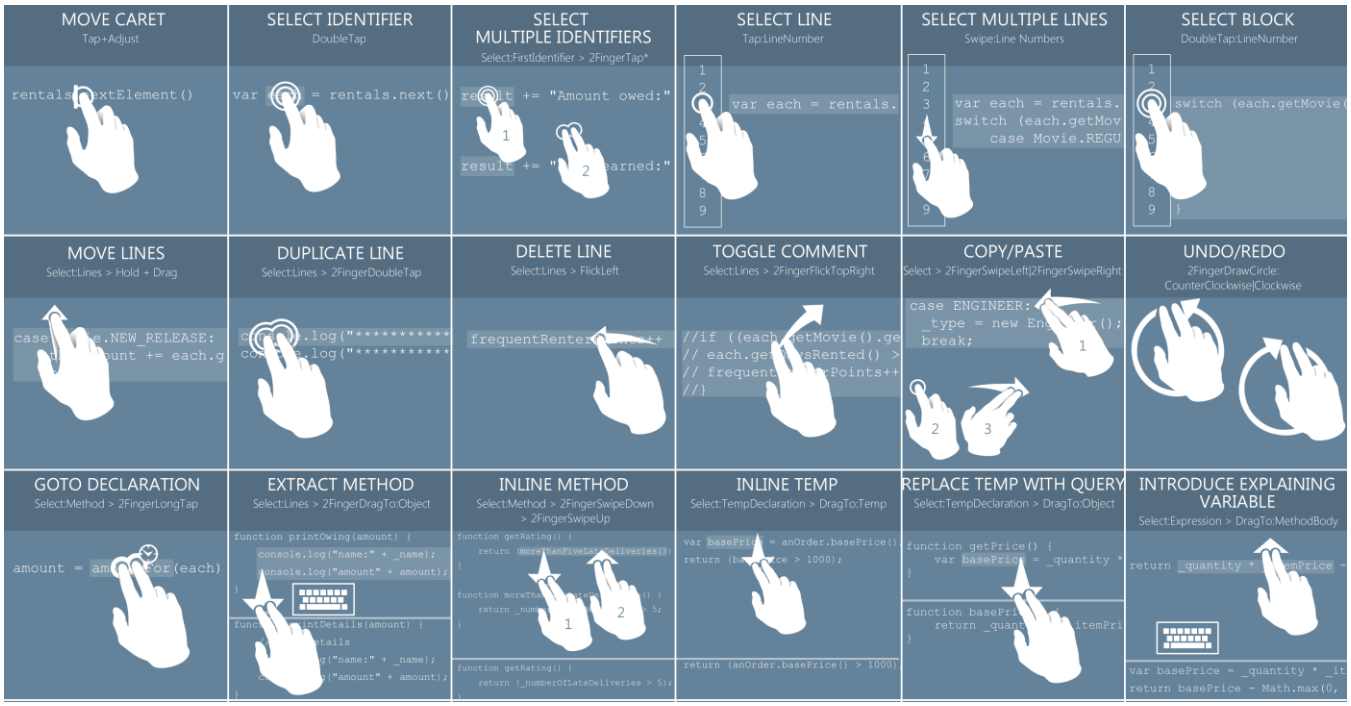

Figure 4: Gesture set for basic selection, editing and refactoring operations for text-based editors on touchscreens. (*SelectFirstIdentifier > 2FingerTap* * means: Select the first identifier, then (>) perform multiple (*) taps using 2 fingers.)

usability aspect the study revealed the need for additional "buffer zones" at the top and bottom of the editor area: Almost all participants touched buttons in the navigation bar by accident when they tried to perform their gestures in the editor area.

DISCUSSION

Our current work can be extended in several directions. First, we propose gestures only for a basic subset of commands. Integration of more functionality requires additional selection triggers since not all commands can easily be mapped to gestures. Second, the code examples in our tasks included only "intra-file" source code. It remains open how certain commands would best work with multiple files. Third, some of the common refactoring tasks need additional configuration or user input with the keyboard. Our current command set, however, focuses on the interaction used to trigger the command. Fourth, the lab setting might have prevented users from using two-handed interaction since the tablet could not be picked up by participants to freely interact with the test system. Finally, our work does not address the problem of entering large amounts of new code and still relies on existing on-screen keyboards.

CONCLUSION

With the continuing adoption of touchscreens and mobile devices, it seems logical that development tools need to be optimized for multi-touch and gestural interaction in the future. In addition, approaches such as visual programming have led to interesting concepts but have not gained much acceptance among professional programmers. This might partly be due to the fact that developers wish to keep working with programming languages and tools they have become experienced in over the years. Therefore, rather than radically changing development tools, we suggest to enhance existing text-based code editors with gestural interaction for basic selection and edit operations. This work presents a test setup that involves users to find suitable ways of interacting with source code on touchscreens and proposes design recommendations for implementers of touch-enabled development environments.

REFERENCES

1. Bragdon, A., Zeleznik, R., Reiss, S. P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., LaViola, J. J. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proc CHI '10*, 2503-2512.
2. Bragdon, A., DeLine, R., Hinckley, K., Morris, M. R., CodeSpace: Touch + Air Gesture Hybrid Interactions for Supporting Developer Meetings. In *Proc ITS'11*, 212-221.
3. DeLine, R., Rowan, K. Code Canvas: Zooming towards Better Development Environments. In *Proc ICSE'10*, 207-210.
4. DeLine, R., Bragdon, A., Rowan, K., Jacobsen, J., Reiss, S. P. Debugger canvas: industrial experience with the code bubbles paradigm. In *Proc ICSE'12*, 1064-1073.
5. Hesenius, M., Medina, C. D. O., Herzberg, D. Touching Factor: Software Development on Tablets. In *Lecture Notes in Computer Science 7306 (2012)*, 148-161.
6. Kim, M., Zimmermann, T., Nagappan, N. A Field Study of Refactoring Benefits and Practice. In *Proc. FSE'12*.
7. Knaus, C. Interaction design for software engineering: boost into programming future. In *Interactions* 15, 4 (July 2008), 71-74. http://doi.acm.org/10.1145/1374489.1374508
8. Ko, A., Aung, H. H., Myers, B. Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks. In *Proc. ICSE '05*, 126-135.
9. Negara, S., Chen, N., Vakilian, M., Johnson, R. E., Dig, D. Using Continuous Change Analysis to Understand the Practice of Refactoring. *Technical Report*, http://hdl.handle.net/2142/30759.
10. Parnin, C., Görg, C., Rugaber, S. CodePad: Interactive Spaces for Maintaining Concentration in Programming Environments. In *Proc. SOFTVIS'10*, 15-24.
11. Sauro, J., Dumas, J. S. Comparison of Three One-Question, Post-Task Usability Questionnaires. In *Proc. CHI'09*, 1599-1608
12. Shatnawi, R., Li, W. An Empirical Assessment of Refactoring Impact on Software Quality Using a Hierarchical Quality Model. *International Journal of Software Engineering and Its Applications* 5, 4 (2011), 127-149.
13. Tillmann, N., Moskal, M., de Halleux, J. TouchDevelop: programming cloud-connected mobile devices via touchscreen. In *Proc. ONWARD'11*, 49-60.
14. Vakilian, M., Chen, N., Negara, S., Rajkumar, B. A., Bailey, B. P., Johnson, R. E. Use, disuse, and misuse of automated refactorings. In *Proc. ICSE'12*, 233-243.
15. Wigdor, D., Wixon, D. *Brave NUI World: Designing Natural User Interfaces for Touch and Gesture*, Morgan Kaufmann, Burlington, MA, USA, 2011.
16. Wobbrock, J. O., Morris, M. R., Wilson, A. D. User-defined gestures for surface computing. In *Proc. CHI'09*, 1083-1092.
17. Xing, Z., Stroulia, E. Refactoring Practice: How it is and How it Should be Supported – An Eclipse Case Study. In *Proc. ICSM'06*, 458-468.