Designing Useful Tools for Developers

Thomas D. LaToza
Institute for Software Research
Carnegie Mellon University
tlatoza@cs.cmu.edu

Brad A. Myers
Human Computer Interaction Institute
Carnegie Mellon University
bam@cs.cmu.edu

Abstract

Designing useful tools for developers requires identifying and understanding an important problem developers face and designing a solution that addresses this problem. This paper describes a design process that uses data to understand problems, design solutions, and evaluate solutions' usefulness.

Categories and Subject Descriptors D.2.6 [**Software Engineering**]: Programming Environments; D.3.0 [**Programming Languages**]: General

General Terms Experimentation, Human Factors

Keywords empirical software engineering, program comprehension, design process

1. Introduction

Is a development tool useful? This question ultimately asks how the tool affects developers' work. This is not, as others argue, a question of philosophy, mathematics, or esthetics [8], but of science: if a developer adopts a tool, is his or her work faster or better? Claims about a tool's usefulness are falsifiable statements about the real world and thus scientific. For the purposes of this paper, the word "tools" includes anything used by a developer to develop software, ranging from development environment plugins to online documentation to type systems to programming languages.

Unfortunately, usefulness is challenging to measure. After designing and implementing a tool's core features, more work may be required before developers can or will adopt the tool such as adding features, fixing usability problems, or even building a user interface. This work is traditionally viewed as an engineering effort with little research value, but generally must be completed before the tool's usefulness can be measured. The most direct measurement of usefulness is through a field deployment. But measuring usefulness in the field requires controlling a myriad of confounding factors. If the developers who adopted a tool fix bugs 10% faster than they did last week, was this effect caused by the tool or by easier bugs, new debugging strategies, or more code knowledge? Even when confounds have been controlled, skeptics may still ask if the result generalizes to developers with different skills, in different domains, with different processes, or with different existing tools. While not impossible (c.f., [2][16]), field evaluations are no small undertaking.

Thus usefulness is most often evaluated in the lab. Developers, or (more typically) students with development experience, are brought in and asked to complete tasks while using a new tool versus a comparable existing tool. Developer's performance is measured by recording time and success, and compared between those using the new tool and the control. If differences are statistically significant, the tool's benefits have been demonstrated.

But such a study does not, by itself, demonstrate usefulness. First, does it generalize? Is the result specific to the situation studied; how might it change with different tasks, codebases, or expertise? Second, what does it mean for developers in the field? How frequently do developers do tasks equivalent to those in the study? Do limitations of the tool prevent it from being used in less controlled settings? Third, how did learning a new tool influence the result? If the results showed the tool did not help, was this because the tool is not useful or because developers had not yet learned new strategies or processes [7]?

Due to these challenges, tool designers often evaluate usefulness less empirically with a motivating example. Motivating examples demonstrate the tool on an example task and can be used to claim that while existing tools make the task time-consuming, tedious, or error-prone, the tool solves these problems (e.g., statically prevents null pointer exceptions, reduces boilerplate code). But while motivating examples can be highly effective for explaining a tool's features and usage, they do not show that developers do

PLATEAU'11, October 24, 2011, Portland, Oregon, USA.

these tasks, that developers do them as described, that developers have the assumed problems, that developers would use the tool in the way described, and that developers would be more productive if they did so. Motivating examples hypothesize a mechanism by which a tool helps, but do not provide evidence of its existence.

This paper describes a process for using data to design useful tools for developers. Data is used before, during, and after design to understand developer's work and how it is affected by a tool. Exploratory studies generate data to identify and describe a problem. Tools are designed to address specific problems, with lightweight evaluation studies testing early design ideas before a large commitment has been made. Evaluation studies help both quantify performance effects and to understand how these effects occurred, allowing greater generalizability. While this design process is heavily influenced by contextual design [6], this paper explains how it can be adapting to designing tools for developers.

This paper begins by examining the structure of software development work and how tools may support work. The design process is then traced from beginning to end, describing techniques for understanding problems, designing a solution, and evaluating its effects.

2. Supporting development work

Useful tools support software development work. But what is work, and how do tools support it? Software development work can be hierarchically decomposed into tasks through task analysis [10] (see Figure 1). In each task at each level in the decomposition, developers have a goal, either a question to answer or something to accomplish. At the highest level, tasks reflect *activities* – e.g., fixing bugs, implementing features, refactoring. These can be decomposed into *sub-activities* – e.g., debugging, editing code, reusing code, understanding code. At a lower level, developers formulate a specific plan for accomplishing a goal as a strategy – a sequence of steps. And, steps in a strategy may themselves involve using other strategies.

Tasks are driven by a goal. For many of developers' activities, the goal is to answer a question. Developers experience problems answering a question when strategies to answer it are time consuming, error-prone, or tedious with the available tools [1][3][5][11][12][17][18][19][20][21][23][24][25][26][27][28][31][34][35]. Debugging, determining how to reuse an API, or predicting the implications of a change are all examples of problems with answering questions. Tools support answering questions by helping developers answer them more quickly or accurately, often by automatically producing information. Debuggers, type systems, reverse engineering and understanding tools, defect detectors, and protocol miners all provide information intended to help answer questions. For the designer, the key challenge is to determine exactly what question developers ask and what information is sufficient to answer it.

Figure 1. Developers' work is hierarchically composed of tasks that may be activities, strategies, or steps. All tasks have a goal and may also have associated problems.

Other activities involve accomplishing something. Here, developers seek to change an artifact in some fashion. Like answering questions, strategies can again be problematic when they are time-consuming, error-prone, or tedious.

Tools support work by making a strategy faster or more successful. For example, the WhyLine [18] helps developers debug (an activity), which involves answering why and why not questions about the causes of erroneous output (questions). Rather than formulate and test hypotheses about a bug's cause (a frequently unsuccessful strategy), the WhyLine lets developers directly select output and follow dynamic slices explaining why it did or did not occur (a new strategy made possible by the WhyLine). Studies of the WhyLine provided evidence of its usefulness by demonstrating that developers frequently ask why and why not questions during debugging, that many of the hypotheses developers formulate are wrong [23], that developers can use the WhyLine to answer their why and why not questions, and that the WhyLine helps developers work quantitatively more effectively [19]. Together, these studies provide a theory describing how the WhyLine supports work.

3. Understanding a problem

Useful tools solve an *important* problem. Problems may be important for many reasons, but often not the ones researchers expect. Exploratory studies can help to identify and understand a problem's true cause and importance.

3.1. Important problems

Problems can be characterized along three dimensions: frequency, duration, and quality impact. Problems vary along all these dimensions and come in many shapes and sizes. Problems need not be frequent and long to be important: an hour every week may have the same direct impact on productivity as 30 seconds 120 times a week. An autocomplete tool which frequently saves a second of time

might have the same impact as a specification checker preventing a very hard to debug but infrequently occurring defect. Tools may also indirectly impact productivity; perhaps the autocomplete tool helps keep the developer more focused on the task, reducing time to switch among tasks. Problems that are neither frequent nor long in duration can be important if they substantially impact quality.

Useful tools must solve an important problem in order to justify their adoption cost. Adopting a tool imposes costs to install it and to learn how to use it effectively, and there is always the risk that it will not actually help. One reason practicing developers are skeptical of academic tools is that their perceived benefits are too small [14]. Therefore, determining which problems are sufficiently important to matter, so the tools addressing those problems will be perceived to be beneficial, is an important function of exploratory studies.

Solving a problem requires understanding its true cause, which often requires digging into symptoms. Consider the problem of code duplication. Developers have long been accused of copy and paste reuse – reusing short snippets of code by copying and editing rather than refactoring code into new abstractions. Copy and paste reuse creates clones, which are frequent in open source codebases (c.f., [3]). To solve this problem, researchers designed tools to detect short copy and paste clones, expecting developers to use these tools to find and refactor clones [17]. But other researchers believe that code duplication is not caused by a lack of awareness of clone's presence at all, but by expressiveness – existing languages make refactoring clones to abstractions mentally challenging, introduce unnecessary overhead, make code more complex, and may not support all types of clones [36]. Other studies suggest that copy and paste reuse is but one of many causes of code duplication, with others such as forking codebases for organizational reasons and maintaining old versions [27]. As a result, commercial clone detectors have found more success focusing on a different problem. For example, developers sometimes work with code where entire codebases have been duplicated and modified, in order to maintain different versions, configurations, or releases. Fixing a bug then requires an extra step: find the equivalent code in all of the codebases. Missing a copy of the bug can be a serious problem: nearly half of software releases contain a security vulnerability already fixed elsewhere in the codebase [32]. Pattern Insight's clone detector is primarily used to solve this problem. Common wisdom can suggest interesting aspects of software development, but there is often much more to understand.

One of the most successful approaches to identifying and understanding a problem is to identify problems with using a strategy or answering a question (c.f., [20][24][34][21][35][12][1]). Problems at this level can be directly addressed by tools. For example, one study examined how developers choose a class in an API to accomplish a goal [35]. Developers pick a candidate, try it out by instantiating it, and often get compiler errors prompting them to supply a required parameter. Developers then investigate how to correctly construct the class, only to later discover that the class is missing the methods they need. The compiler errors encourage a *premature commitment*, causing investigation into something that may be irrelevant. Understanding this strategy (how developers pick classes in an unfamiliar API) helps to identify the problems that make it hard and time consuming (compiler errors encourage premature commitment). Tools can then be designed to address these problems (preventing premature commitment) and evaluated in terms of their success in doing so.

3.2 Choosing an exploratory study

Exploratory studies help to identify important problems and understand their cause. Exploratory studies may gather data both through developers' perceptions of problems and through directly examining the problems themselves. Developer perception can be an important source of ideas, particularly for poorly understood tasks. Developer may identify challenging tasks or hard-to-answer questions. But other problems are less salient, and may be important without developers realizing it. For example, developers may not notice how much time they spend scrolling to revisit code [20].

One of the easiest exploratory studies to conduct is an interview. Interviewing developers can reveal a developers' typical tasks and problems (c.f., [27][15]). But recollection of the past is imperfect: people give vague descriptions and generalize [6]. While generalizations may suggest ideas, they may not be based on facts and can be biased by opinion and perception. But, for tasks or situations for which little is known, interviews give a sense of the basics and help to focus further study on interesting aspects.

Contextual inquiries augment interviews with direct observations, using a real, in-the-moment tasks to provide context [6]. Contextual inquiries replace generalization with examples; an experimenter watches the developers as they work and asks questions about the task at hand. Developers work on a representative task and think-aloud as they work. When the developer's goals, questions, or strategy is unclear, the experimenter asks for clarification. When the developer generalizes, the experimenter asks for a concrete example. When frequently interrupting is inconvenient or obtaining accurate timing of steps and activities is important, direct observations can be used alone, without an embedded interview. The experimenter may still briefly interrupt, but interruptions are focused on brief clarifications rather than extended discussions. Direct observations can be conducted both as field studies (c.f., [21][24][34]), watching developers in the field do their everyday work, or in lab studies which permit choice of the task and compari-

sons of developers doing the same work (c.f., [24][20][12][35][1]).

Direct observations lead to generalization by analyzing the data afterwards. Depending on the aspect or situation of interest, many types of analysis are possible. Simply reporting observations is sometimes sufficient. But often it helps to examine their generality by looking for patterns, often through content analysis [33]. Taxonomies investigate what things exist in the data – e.g., types of activities, questions, and strategies.

3.3. Understanding context and frequency

How well a strategy works often depends on the context [25]. Consider answering the question "Does this method repaint the screen?" One strategy which developers use is to determine, through code inspection and by traversing paths through the code, if the code calls repaint. Information foraging predicts that developers use their knowledge to pick which edge to traverse based on its similarity to the goal [28]. This is sometimes easy. But when there are many edges to choose from, longer paths to follow, or identifiers that are misleading, this strategy is likely to take longer or fail. Other factors that influence its difficulty include characteristics of the code and the developer. For example, developers less knowledgeable about the code may have a harder time predicting which identifiers are most related to what they're trying to find. Learning about factors influencing a strategy's success is an important part of understanding a problem, and helps ensure that solutions can be targeted to the most important situations.

While observations and interviews help to understand questions, strategies, and problems, data about frequency is limited by the few situations observed. Frequency can be measured through studies designed to sample many developers such as surveys or indirect observations. Surveys gather frequency data by asking many developers questions (c.f., [27][24][21][5]). Surveys can also be used to understand correlations. For example, one study found that a class of questions becomes neither less frequent nor easier to answer as developers become more experienced or spend more time in a codebase [24]. Indirect observations gather data about developer's work not by directly seeing it but by capturing summary data, such as with logging, or by studying artifacts created by work such as code, code change logs, emails, bug discussions, and forum posts (c.f., [22][4][30]). For example, one study measured the prevalence of protocols through an automated technique for detecting protocols in code [4], built a taxonomy of protocol types, and examined their typical complexity.

4. Designing a solution

Designing a tool begins with an important problem to solve. A problem is the beginning of a solution – it identifies a specific aspect of work to improve. Designing a solution envisions a new way of doing this work and determines the features necessary to make this possible. Designing a tool is a leap from an old to a new way of working; designing a tool that solves a problem is an inherently creative process that no amount of data can guarantee. But data can help to understand what a design must achieve to solve a problem and understand if it is likely to succeed.

When using a tool to answer a question, a developer must translate a high-level question (e.g., what caused this bug?) into lower level questions the tool supports (e.g., using a breakpoint to answer, "What's the value of this expression when this code executes?"). For the designer, the key challenge is ensuring that the information that the tool provides really helps to answer the high-level question more quickly or accurately than the alternatives. One way to bridge this gulf is to understand how developers currently work: what strategies do they use to answer high-level questions, and what lower-level questions do these strategies entail? For example, we found that developers sometimes answer questions about the implications of a change [34] (e.g., what it might break) by searching along control flow for things that the code does [24]. Helping developers search along control flow is likely to help developers answer higher-level implication questions, as developers are already using this strategy. But supporting developers' current strategy is not the only approach – tools could instead provide an entirely new strategy that is impossible with existing tools. But it is then necessary to determine if the low-level questions the tool answers actually help answer higher-level questions, whether these low-level questions are really an important part of the problem, and whether the developers will think to use the strategy the tool supports in the relevant situations.

Consider the following example: many automated debugging tools attempt to predict the faulty statement that caused the bug and provide the developer a ranked list of candidate statements. These tools change how developers answer "What caused this bug?" by letting developers inspect the list of statements and answer the question "Which statement contains the fault?" But how big a part is finding the faulty statement in determining the cause of a bug? Is seeing a statement sufficient for a developer to determine that it is faulty? A user study investigated this question by comparing automated to conventional debugging tools [31]. It suggests the answer is no: most developers spend an average of 10 minutes inspecting each statement to understand how it might have caused the bug. As a result, the automated debugging tools only helped a fraction of developers debug one of two tasks more quickly. Understanding exactly what information a tool should provide to help answer a high-level question is crucial to a tool's success.

5. Evaluating a solution

A tool is the embodiment of a tool designer's assumptions about how developers currently work and the way in which that work may be more effectively supported. Unfortunately for the designer, these assumptions may be wrong. This risk can be minimized by getting feedback early in the design process through *prototypes* and *lightweight evaluation studies*. In a *paper prototype study* [9], users interact with screenshots, narrating which buttons they click, while the experimenter manually simulates the tool by showing the next screen. Higher fidelity mockups are also possible. In a Wizard of Oz study [29], an interface is built, but the implementation is remote-controlled by the experimenter. For example, a bug detector might provide error messages that seem to be automatically generated when they are actually triggered by the experimenter. Such a study allows the effects of different error message designs to be evaluated before determining how such errors will be generated. Lightweight evaluations studies enable an iterative design process in which is tailored to what works rather than what the designer assumes will work.

The usefulness of a tool depends both on its success in solving a problem and supporting work (mechanism) and the importance of the underlying problem. Lab studies are most effective for understanding mechanism. Do developers use the tool to answer questions? Does the tool help them do it more quickly or successfully than before? What strategy(s) does it support? On what aspects of the situation does the strategy depend? How might these aspects effect how developers use the tool? Did developers enjoy using the tool? Evaluating a tool's usefulness is difficult without understanding *how* it supports work.

Consider an example: one study investigated the productivity effects of dynamic typing [13]. Participants took anywhere from 4% to 42% less time using a dynamically typed language compared to an otherwise identical statically typed language. But does this result generalize? Did the tasks involve any situations in which static typing might be expected to provide useful feedback? If so, how did developers use this information and why did it not help? Was it simply that the error messages provided were poorly designed and unhelpful? What were developers doing during the additional time in the statically typed condition? Would developers working with different codebase or with different tools still have these problems? Are there ways to provide useful static typing feedback without incurring the productivity costs? Answering these questions requires a deeper understanding of the tasks, activities, questions, and strategies developers did.

Lab studies can also quantitatively measure a tool's effect on task time and success. A result which shows an effect is best viewed as an existence proof: it is possible to achieve significant productivity benefits. Such results are an important demonstration that a tool can have a strong effect (or that it did not) (c.f., [26][19][31][11][13]). Yet skeptics can always argue that, for a slightly different task or situation, the tool's benefits may vanish. Studying a tool in more situations helps to address these concerns. But the strongest argument is also based on mechanism – an explanation of *how* a tool changes the way that developers work. Mechanisms predict the effects of a tool in unobserved circumstances. These predictions may be wrong, but even then, they focus research on what prevented the predicted effects from occurring. Did developers not do the expected tasks, use unexpected strategies, or did they experience unexpected problems with these strategies?

Another benefit of understanding mechanism is in designing lab study evaluations. Lab studies sample particular situations by picking tasks, participants, and materials. Which of the many possible situations are most interesting to study? Situations where developers do not do the task supported by the tool are not interesting, as the results only show the tool is not relevant. However, observing tasks for which the benefits are unclear is interesting. Situations where the tool is expected to have a strong effect provide evidence that the effect actually exists.

6. Conclusions

Designing a useful tool requires more than finding a compelling motivating example, evaluating the tool's technical merits, and performing a carefully designed user study. Designing a useful tool requires understanding how a tool supports work and addresses an important problem that developers face. This understanding is built over time through hypotheses and studies, investigations of the problem and potential solutions, and through direct and indirect observation. Understanding involves identifying both the *mechanism* of how it supports tasks, questions, and strategies and the *frequency* that such situations occur. And evaluations must examine not only whether some quantitative productivity effect is possible but how, when, and why this effect is achieved.

This paper has examined individual theories in their role in understanding tools. But, as theories grow in number and sophistication, they may have much in common. This may lead to larger theories of developer's work and to better predictions of usefulness. Theory is a key part of building a science of development tools.

Acknowledgements

We thank Andy Ko for many discussions on research methodology and thank Marwan Abi-Antoun for helpful comments. This research was funded in part by the National Science Foundation under grant CCF-0811610.

References

[1] Abi-Antoun, M., Ammar, N. and LaToza, T. 2010. Questions about object structure during coding activities. *ICSE Workshop on Cooperative and Human Aspects of Software Engineering* (2010).

[2] Atkins, D.L., Ball, T., Graves, T.L. and Mockus, A. 2002. Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor. *Transactions on Software Engineering*. 28, 7 (Jul. 2002), 625–637.

[3] Baker, B.S. 1995. On finding duplication and near-duplication in large software systems. *Working Conference on Reverse Engineering* (1995), 86–95.

[4] Beckman, N.E., Kim, D. and Aldrich, J. 2011. An empirical study of object protocols in the wild. *European Conference on Object-Oriented Programming* (2011).

[5] Begel, A., Phang, K.Y. and Zimmermann, T. 2010. Codebook: discovering and exploiting relationships in software repositories. *ICSE '10*. (Jan. 2010).

[6] Beyer, H. and Holtzblatt, K. 1997. *Contextual Design: Defining Customer-Centered Systems*. Morgan Kaufmann.

[7] Blackwell, A.F. 2002. First steps in programming: a rationale for attention investment models. *Symposia on Human Centric Computing Languages and Environments* (2002), 2–10.

[8] Bracha, G. 2010. The fitness function for programming languages: a matter of taste *Keynote at the Evaluation and Usability of Languages and Tools (PLATEAU) at SPLASH.* (2010).

[9] Buxton, B. 2007. *Sketching user experiences*. Morgan Kaufmann.

[10] Crystal, A. and Ellington, B. 2004. Task analysis and human-computer interaction: approaches, techniques, and levels of analysis. *Americas Conference on Information Systems* (2004).

[11] de Alwis, B., Murphy, G.C. and Robillard, M.P. 2007. A Comparative Study of Three Program Exploration Tools. *International Conference on Program Comprehension* (2007), 103–112.

[12] Fleming, S.D., Kraemer, E., Stirewalt, R.E.K., Xie, S. and Dillon, L.K. 2008. A study of student strategies for the corrective maintenance of concurrent software. *International Conference on Software Engineering* (2008).

[13] Hanenberg, S. 2009. What is the impact of static type systems on programming time *PLATEAU Workshop at OOPSLA '09*. (Oct. 2009).

[14] How do practitioners perceive Software Engineering Research? *http://catenary.wordpress.com/2011/05/19/how-do-practitioners-perceive-software-engineering-research/*. Accessed: 08-01-2011.

[15] Howison, J. and Herbsleb, J.D. 2011. Scientific software production: incentives and collaboration. *Computer Supported Cooperative Work* (2011).

[16] Jaspan, C., Chen, I.-C. and Sharma, A. 2007. Understanding the Value of Program Analysis Tools. *Companion to the Conference on Object-Oriented Programming Systems and Applications* (2007).

[17] Kamiya, T., Kusumoto, S. and Inoue, K. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System. *Transactions on Software Engineering*. 28, 7 (Jul. 2002).

[18] Ko, A.J. and Myers, B.A. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. *International Conference on Software Engineering* (2008), 301–310.

[19] Ko, A.J. and Myers, B.A. 2009. Finding Causes of Program Output with the Java Whyline. *Conference on Human Factors in Computing Systems* (2009), 1569–1578.

[20] Ko, A.J., Aung, H. and Myers, B.A. 2005. Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks. *International Conference on Software Engineering* (2005), 126–135.

[21] Ko, A.J., DeLine, R. and Venolia, G. 2007. Information Needs in Collocated Software Development Teams. *International Conference on Software Engineering* (2007), 344–353.

[22] Ko, A.J., Myers, B.A. and Chau, D.H. 2006. A linguistic analysis of how people describe software problems. *Visual Languages and Human-Centric Computing*. (2006).

[23] Ko, A.J. 2008. *Asking and answering questions about the causes of software behavior*. Dissertation, Carnegie Mellon University.

[24] LaToza, T.D. and Myers, B.A. 2010. Developers Ask Reachability Questions. *International Conference on Software Engineering* (2010), 185–194.

[25] LaToza, T.D. and Myers, B.A. 2010. On the importance of understanding the strategies the developers use. *ICSE Workshop on Cooperative and Human Aspects of Software Engineering* (2010).

[26] LaToza, T.D. and Myers, B.A. 2011. Visualizing Call Graphs. *Visual Languages and Human-Centric Computing* (2011).

[27] LaToza, T.D., Venolia, G. and DeLine, R. 2006. Maintaining Mental Models: A Study of Developer Work Habits. *International Conference on Software Engineering* (2006), 492–501.

[28] Lawrance, J., Bogart, C., Burnett, M., Bellamy, R., Rector, K. and Fleming, S. 2010. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *IEEE Transactions on Software Engineering*. 99 (2010).

[29] Maulsby, D., Greenberg, S. and Mander, R. 1993. Prototyping an intelligent agent through Wizard of Oz. *Conference on Human Factors in Computing Systems* (1993), 277–284.

[30] Murphy-Hill, E., Parnin, C. and Black, A.P. 2009. How we refactor, and how we know it. *International Conference on Software Engineering* (2009), 287–297.

[31] Parnin, C. and Orso, A. 2011. Are Automated Debugging Techniques Actually Helping Developers *International Symposium on Software Testing and Analyisis* (2011), 199–209.

[32] Pattern Insight. *http://patterninsight.com*. Accessed: 10-05-2011.

[33] Patton, M.Q. 2002. *Qualitative research and evaluation methods*. Sage Publications.

[34] Sillito, J., Murphy, G.C. and de Volder, K. 2008. Asking and Answering Questions during a Programming Change Task. *IEEE Trans. Softw. Eng* 34, (Jul. 2008), 434–451.

[35] Stylos, J. and Clarke, S. 2007. Usability implications of requiring parameters in objects' constructors. *International Conference on Software Engineering*. (2007), 529–539.

[36] Toomim, M., Begel, A. and Graham, S.L. 2004. Managing Duplicated Code with Linked-Editing. *Symposium on Visual Languages and Human-Centric Computing* (2004).