Report from Dagstuhl Seminar 15222

Human-Centric Development of Software Tools

Edited by

Andrew J. Ko[1], Shriram Krishnamurthi[2], Gail Murphy[3], and Janet Siegmund[4]

1 **University of Washington, USA,** `ajko@uw.edu`
2 **Brown University, USA,** `sk@brown.edu`
3 **University of British Columbia, Canada,** `murphy@cs.ubc.ca`
4 **University of Passau, Germany,** `Janet.Siegmund@uni-passau.de`

Abstract

Over two and half days, over 30 participants engaged in inventing and evaluating programming and software engineering tools from a human rather than tool perspective. We discussed methods, theories, recruitment, research questions, and community issues such as methods training and reviewing. This report is a summary of the key insights generated in the workshop.

Seminar May 25–28, 2015 – http://www.dagstuhl.de/15222
1998 ACM Subject Classification H.5.2 User Interfaces, D.2.3 Coding Tools and Techniques
Keywords and phrases Software engineering, developer tools, human factors
Digital Object Identifier 10.4230/DagRep.5.5.115

1 Executive Summary

Andrew J. Ko

Across our many sessions, we discussed many central issues related to research on the design of human-centric developer tools. In this summary, we discuss the key insights from each of these areas, and actionable next steps for maturing the field of human-centered developer tools.

Key Insights

Theories

Theories are a hugely important but underused aspect of our research. They help us start with an explanation, they help us explain and interpret the data we get, they help us relate our findings to others findings, and they give us vocabulary and concepts to help us organize our thinking about a phenomenon.

There are many relevant theories that we should be using:

- Attention investment is helpful in explaining why people choose to engage in programming.
- Information foraging theory helps explain where people choose to look for relevant information in code.
- Community of practice theory helps us explain how people choose to develop skills over time.

There are useful methods for generating theories, including grounded theory and participatory design. Both can result in explanations of phenomena. That said, there are often already theories about things and we don't need to engage in creating our own.

While theories are the pinnacle of knowledge, there's plenty of room for "useful knowledge" that helps us ultimately create and refine better theories. Much of the research we do now generates this useful knowledge and will eventually lead to more useful theories.

Study Recruitment

Whether developers agree to participate in a study depends on several factors:

- One factor is how much value developers perceive in participating. Value might be tangible (a gift card, a bottle of champagne), or personal (learning something from participation, or getting to share their opinion about something they are passionate about).
- Another factor in recruitment is whether the requestor is part of the developer in-group (e.g, being part of their organization, having a representative from their community conduct the research or recruit on your behalf, become part of their community before asking for their efforts)
- The cost of participating obviously has to be low, or at least low enough to account for the benefit. With these factors in mind, there are a wide range of clever and effective ways to recruit participants:

Monitor for changes in bug databases and gather data at the moment the event occurs. This makes the request timely and minimizes the cost of recall.

- Find naturalistic captures of people doing software engineering work (such as tutorials, walkthroughs, and other recorded content that developers create). This costs the nothing.
- Perform self-ethnographies or diary studies. This has some validity issues, but provides a rich source of data.
- Tag your own development work through commits to gather interesting episodes.
- Find where developers are and interview them there (e.g., the Microsoft bus stop, developer conferences), and generate low-cost, high-value ways of getting their attention (and data).

Research Questions

There was much discussion of research questions at the conference and what makes a good one. There was much agreement that our questions should be more grounded in theories, so that we can better build upon each others' work.

Many researchers also find that the human-centered empirical studies produce results that are not directly meaningful or actionable to others. There are many possible reasons for this:

- We often don't choose research questions with more than one plausible outcome.
- We often don't report our results in a way that creates conflict and suspense. We need to show readers that there are many possible outcomes.
- We often ask "whether" questions, rather than "why" or "when" questions about tools, leading to limited, binary results, rather than richer, more subtle contributions.

Some of our research questions have validity issues that make them problematic:

- Research questions often fail to understand the populations they are asking about.
- Research questions often get involved in designing tools for people who are already designing tools for themselves. Instead, researchers should be building tools that have never existed, not building better versions of tools that already exist.

One opportunity for collaboration with researchers who are less human-centered is to collaborate on formative research that shapes the direction of research and discover new research opportunities for the field. This may create more positive perceptions of our skills, impact, and relevance to the broader fields of PL and SE.

Human-Centeredness

Historically, HCI concerns have focused on end user experiences rather than developer experiences, but HCI researchers have increasingly focused on developers and developer tools. But HCI often doesn't consider the culture and context of software engineering, and doesn't address the longitudinal / long term factors in education and skill acquisition, and so HCI may not be a sufficient lens through which to understand software engineering.

There is also a need to address low-end developers, not just "experts". Future research topics include the understand learnability of APIs, how to understand the experiences of engineers (from a sociological perspective studies such as Bucciarelli), how to think about tools from a knowledge prerequisite perspective.

Developer Knowledge Modeling

Much of what makes a developer effective is the knowledge in their mind, but we know little about what this knowledge is, how developers acquire it, how to measure and model it, and how to use these models to improve tools or enable new categories of tools. There are many open opportunities in this space that could lead to powerful new understandings about software engineering expertise and powerful new tools to support software engineering. Much of this new work can leverage research in education and learning sciences to get measures of knowledge.

Leveraging Software Development Analytics

We identified identifying different types of data that might be collected on programming processes and products. These included editing activities, compilation attempts and errors, execution attempts and errors, and check-ins. We considered ways in which these data could be enlisted to help improve teaching and learning, as well as the software development process:

- Automated interventions to improve programming processes
- Present visually to aid in decision making
- Generate notifications that could inform learners, teachers, and software developers of key events.
- Generating social recommendations.

These opportunities raise several questions:

- How do we leverage data to intervene in educational and collaborative software development settings?
- How do we design visual analytics environment to aid in decision making?
- Should interventions be automated, semi-automated, or manual? What are the trade offs?

Error Messages

We identified 5 broad classes of errors: (1) syntactic (conformance to a grammar), (2) type, (3) run-time (safety checks in a run-time system, such as array bounds, division by zero, etc.), (4) semantic (logical errors that aren't run-time errors) (5) stylistic. We distinguished between errors and more general forms of feedback, acknowledging that both needed support; in particular, each of these could leverage some common presentation guidelines.

We discussed why research has tended to focus more on errors for beginners than feedback for developers. Issues raised included the different scales of problems to diagnose across the two cases and differences in social norms around asking for help from other people (developers might be less likely to ask other people for help in order to protect their professional reputations). We discussed whether tools should report all errors or just some of them, and whether tools should try to prioritize among errors when presenting them. These had different nuances in each of students and practicing developers. We discussed the example of the coverity tool presenting only a subset of errors, since presenting all of them might lead developers to reject the tool for finding too much fault in the their code.

We discussed and articulated several principles of presenting errors: (1) use different visual patterns to distinguish different kinds of errors; (2) don't mislead users by giving incorrect advice on how to fix an error; (3) use multi-dimensional or multi-modal techniques to revelt error details incrementally; (4) when possible, allow programs to fail gently in the face of an error (example: soft typing moved type errors into run-time errors that only tripped when a concrete input triggered the error – this gives the programmer some control over when to engage with the error after it arises); (5) consider ways to allow the user to query the system to narrow down the cause of the error (rather than require them to debug the entire program).

There are several open research questions:

- Should error and feedback systems become interactive, asking the user questions to help diagnose a more concrete error (rather than report a more abstract one, as often happens with compiler syntax errors)?
- Can grammars be tailored to domain-specific knowledge to yield more descriptive error messages?
- Can patterns of variable names be used to enforce conventions and reduce the rates of some kinds of errors?
- At what point should error systems expect the user to consult with another human, rather than rely only on the computer.
- When is it more helpful to show all errors (assuming we can even compute that) versus a selection of errors? How much detail should be presented about an error at first? Does presenting all information discourage users from reading error messages?

Reviewing

Researchers in human aspects of software engineering feel a strong sense of hostility towards human-centered research, despite some recent successes in some software engineering venues. Reasons for this hostility include:

- Many human-centered researchers evaluate and critique tools without offering constructive directions forward. This creates a perception that human-centered researchers dislike or hate the research that others are doing.
- Many human-centered researchers are focused on producing understanding, whereas other researchers are focused on producing better tools. This goal mismatch causes reviewers to apply inappropriate criteria to the importance and value of research contributions.
- Many research communities in programming languages and software engineering still lack sufficient methodological expertise to properly evaluate human-centered empirical work.

- It's not explicit in reviews whether someone's methodological expertise is a good match for a paper. Expert it in a topic, not expert in a method. This leads to topic expertise matches without methodological expertise matches.
- Many challenges in reviewing come from the difference between judging a paper's validity versus judging how interesting a paper is. Non-human centered researchers do not often often find our questions interesting.

We are often our own worst enemies in reviews. We often reject each other because we're too rigid about methods (e.g., rejecting papers because of missing interrater reliability). On the other hand, we have to maintain standards. There's a lot of room for creativity in establishing rigor that is satisfying to reviewers, and we should allow for these creative ways of validating and verifying our interpretations.

Methods Training

Empirical methods are not popular to learn. However, when our students and colleagues decide to learn them, there are many papers, textbooks, classes and workshops for learning some basic concepts in human-subjects software engineering research.

There are many strategies we might employ to broadly increase methodological expertise in our research communities:

- We should spend more time in workshops and conferences teaching each other how to do methods well.
- Software engineers need to learn empirical methods too, and teaching them as undergraduates will lead to increased literacy in graduate students.
- There is much we can do to consolidate and share teaching resources that would make this instruction much more efficient.
- HCI research methods are broadly applicable and there are many more places to learn them.

There aren't good methods for researching learning issues yet. Moreover, most of these methods cannot be learned quickly. We must devise ways of teaching these methods to students and researchers over long periods of time.

2 Table of Contents

3 Poster Abstracts

3.1 Teaching Usability of Programming Languages

Alan Blackwell (University of Cambridge, GB)

Main reference Usability of Programming Languages. Lecture notes for MPhil in Advanced Computer Science, University of Cambridge Computer Laboratory
URL http://www.cl.cam.ac.uk/teaching/1415/P201/p201-lecturenotes-2015.pdf

I presented an overview of my graduate course teaching usability of programming languages. The course itself is documented here: http://www.cl.cam.ac.uk/teaching/1415/P201/. Lecture notes for the course are in the bibliographic reference.

3.2 The GenderMag Kit: To Find Usability Issues from a Gender Perspective

Margaret M. Burnett (Oregon State University, US)

URL http://eusesconsortium.org/gender

The way people use software features often differs according to their gender, especially when the software is for problem-solving, as when figuring out budgets, or understanding visualizations, or debugging. However, many software features are inadvertently designed around the way males tend to use software.

How using GenderMag helps make software gender-inclusive:

The GenderMag method helps software developers and usability professionals identify features that don't take females' common usage patterns into account. The GenderMag method can also be used to find features that don't take males' common usage patterns into account.

- The GenderMag method encapsulates how five facets of gender differences (motivations for use, information processing style, computer self-efficacy, attitude toward risk, and willingness to explore/tinker) affect the ways males and females tend to use software.
- Software developers and usability professionals use the method to find gender-inclusiveness issues. They can then fix these issues one at a time, so as to individually take down barriers that may disproportionately affect one gender but could also affect a fraction of the other.
- Researchers who have used early versions of GenderMag in this way have found that their software becomes more gender-inclusive and better liked by its users overall.

GenderMag consists of a set of GenderMag Personas to bring the facets to life, and a GenderMag Cognitive Walkthrough to embed use of the personas in a process.

The GenderMag Personas

- Personas represent "archetypes" of users of a software system. The focus of the male and female GenderMag personas is bringing to life the above five facets of gender differences.

The GenderMag Cognitive Walkthrough

- The GenderMag CW is a gender-specialized Cognitive Walkthrough (CW). A (regular) CW analyzes how easily users new to the system can accomplish specific tasks with that system.
- The GenderMag CW adds to the analysis process explicit use of the above five facets of gender differences and the personas.

No background in gender difference research is needed to use GenderMag. It is intended for any software developer or usability professional interested in identifying the features of their software that may not be gender-inclusive.

3.3 Software Tools and Practices

Yvonne Dittrich (IT University of Copenhagen, DK)

Main reference Y. Dittrich, "What does it mean to use a method? Towards a practice theory for software engineering," Information and Software Technology, 70(Feb. 2015):220–231, 2015.
URL http://dx.doi.org/10.1016/j.infsof.2015.07.001

How tools are used depends on how they are embedded in specific practices. Tools often take specific processes and methods for given. However, as we all know, the use of the same tool does not necessarily result in comparable impacts on the software development practice. E.g. Damian et al. (2007) report on a case of a team distributed between US and Canada, using the same method and tooling. One issue was the use of the Code Versioning System. Whereas in one locality, the team members relied on the commit comments distributed to the whole team, the team members in the other locality used extra mails distributed via a mailing list to highlight changes affecting other members of the team. Damian et al. attributed the resulting break-downs to cultural differences. Culture here refers to locally shared but across sides diverging practices. (See also Agar 1996, p. 9). But what are practices?

A Practice Concept for Software Engineering

In philosophy, the notion of practice has been discussed since the time of ancient Greek philosophy. A historical overview of the development of the practice concept is found in (Schmidt 2014). Schmidt defines practice as 'normative regulated contingent activity' (Schmidt 2014, p. 437), arguing that the modern concept of practice focuses 'on the ways in which the competent actor in his or her action is taking the particular conditions into account while committed to and guided by the appropriate general principles ('theory', 'rules')' (Schmidt 2014, p. 436). Wittgenstein's Philosophical Investigations have inspired a number of contemporary schools in social science and philosophy. In his introduction to practice theory, Nicolini (2012) talks about practice theories. To ground a practice theory in a coherent and consistent manner, I refer in (Dittrich 2015) to Schatzki's 'Social Practice. A Wittgensteinian approach to human activity and the social' (1996) as well as Knorr Cetina's article 'Objectual Practice' (2001): Schatzki defines, based on Wittgenstein, practice as a "…temporally unfolding and spatially dispersed nexus of doings and sayings. (…) to say that the doings and sayings forming a nexus is to say that they are linked in certain ways. Three major avenues of linkage are involved: 1) through understandings, for example, of what to say and do; (2) through explicit rules, principles, precepts, and instructions;

and (3) through what I will call teleoaffective structures embracing ends, projects, tasks, purposes, beliefs, emotions, and moods.” [12, p. 89] Such practices take place in specific settings making use of tools, materials and objects that acquire their meaning through the practice they are supporting (Schatzki 1996, p. 113-114). Based on Knorr Cetina’s concept of epistemic practices (2001), I argue that software development is an epistemic practice that unfolds its object and, with it, its own practice as the team proceeds in the development. Such adaptation of practices has been observed as meta and articulation work (Sigfridsson 2010) and as joint tailoring of development environments (Draxler et al. 2014) in software engineering projects. Based on this foundation, methods can be defined as practice patterns, explicitly formulated sets of (tool supported) understandings, rules and teleoaffective structures that need to be integrated in existing practices. See (Dittrich 2015) for a detailed development of the argument.

What does that imply for the use and the usefulness of tools? The usability and usefulness of a tool depends on how it can be embedded into concrete practices. The adaptation of tools can be observed as a conscious and continuous effort of software engineering teams (Draxler et al. 2014: Giuffrida and Dittrich 2015).

Software engineering practices, however, are situated with respect to organisation, domain, and even the specific project. They are not stable but change with the developing understanding of the software product they are meant to develop or evolve. The introduction of a tool implies a development of the practices it is meant to support and needs to be accompanied by meta-work (Strauss 1985), explicit negotiation and agreement of how to embed the tool in the practice. That means that the transfer of tools that are developed based on scientific ideas of programming techniques and languages needs to explicit consider how it relates to industrial practices. Empirical research should focus on the embedment of the new tool into exemplary practices.

References

1 M. H. Agar, The professional stranger: An informal introduction to ethnography. San Diego, CA: Academic Press, 1996.
2 D. Damian, L. Izquierdo, J. Singer, I. Kwan, Awareness in the wild: Why communication breakdowns occur. In International Conference on Global Software Engineering (ICGSE) 2007.
3 S. Draxler, G. Stevens, A. Boden, Keeping the development environment up to date-A Study of the Situated Practices of Appropriating the Eclipse IDE. IEEE Transaction on Software Engineering, 40 (2014), pp. 1061–1074.
4 Y. Dittrich, What does it mean to use a method? Towards a practice theory for software engineering. Information and Software Technology, 2015 in print.
5 R. Giuffrida, Y. Dittrich, A conceptual framework to study the role of communication through social software for coordination in globally-distributed software teams. Information and Software Technology, 63 (2015), pp. 10–30.
6 K. Knorr Cetina, Objectual practice. In: The practice turn in contemporary theory, T. Schatzki, K. Knorr Cetina, and E. von Savigny (eds.), London: Routledge 2001, pp. 175–188.
7 D. Nicolini, Practice theory, work, and organization: an introduction. Oxford University Press, Oxford 2012.
8 T.R. Schatzki, Social practices: A Wittgensteinian approach to human activity and the social. Cambridge University Press 1996.
9 K. Schmidt, The Concept of Practice: What’s the Point?. In COOP 2014-Proceedings of the 11th International Conference on the Design of Cooperative Systems, 27-30 May 2014, Nice (France), Springer Intern. Publ., pp. 427–444.

10 A. Sigfridsson, The purposeful adaptation of practice: an empirical study of distributed software development. PhD Thesis. University of Limerick 2010.
11 A.L. Strauss, Work and the division of labor. Sociol Q, 26(1985), pp. 1–19.
12 L. Wittgenstein, Philosophical Investigations. The German text, with an English translation by G.E.M. Anscombe, P.M.S. Hacker and Joachim Schulte. Revised 4th edition by P.M.S. Hacker and Joachim Schulte. Wiley-Blackwell 2009.

3.4 Programming Languages as Interfaces in Plan Composition

Kathi Fisler (Worcester Polytechnic Institute, US)

This poster explored the ways in which the language constructs that students know affect how they structure solutions to programming problems. The relationship between constructs and solution structure has pedagogic implications: the structures we want students to produce would seem to dictate either the constructs we teach or the problems we assign. This has human-factors implications since some solutions are harder to implement correctly than others. We illustrate the issues with two concrete problems that we have been using in educational studies this year.

3.5 Empirical Support for Contextual Computing Educaiton

Mark Guzdial (Georgia Institute of Technology – Atlanta, US)

Programming is the way it is because of skewed demographics. Changing who programs will dramatically change programming. Changing the language or IDE will do little to change who programs. Humans care about narrative, the context around the activity. People reject the context of programming, not the activity of programming. If we teach programming within context, we broaden the range of people who program.

3.6 Socio-Technical Coordination: How Millions of People use Transparency to Collaborate on Millions of Interdependent Projects on GitHub

James D. Herbsleb (Carnegie Mellon University, US)

Joint work of Dabbish, Laura; Stuart, Colleen; Tsay, Jason; Herbsleb, James D.
Main reference L. Dabbish, C. Stuart, J. Tsay, J. Herbsleb, "Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository," in Proc. of the ACM 2012 Conf. on Computer Supported Cooperative Work (CSCW'12), pp. 1277–1286, ACM, 2012.
URL http://dx.doi.org/10.1145/2145204.2145396

Collectively creating digital things these days often means hoards of people collaborating in open, transparent environments, loosely organized in ecosystems of interdependent projects.

Splitting work across large collections of people has great potential benefits such as tapping a larger talent pool, enabling better matches between tasks and skills, and reducing schedule bottlenecks. But it also gives rise to difficult coordination problems while disabling coordination mechanisms that rely on overarching hierarchies of authority. In this talk, I will develop a socio-technical theory of coordination, and show how colleagues and I empirically validated it in a geographically distributed software development organization. I will show how the theory can be adapted to help interpret the results our qualitative study of coordination practices in GitHub, an open, transparent work environment in which millions of people collaborate on millions of interdependent projects.

3.7 Increasing tool reach through online integration

Reid Holmes (University of Waterloo, CA)

Many development tools have high-quality web interfaces. Software engineering tools can greatly increase their reach by integrating with these interfaces, rather than forcing developers to download and install new tools or to change their development processes by visiting new sites. By injecting content into existing web interfaces, new development tools can be seamlessly integrated into the developer's current tools, increasing the tool's ability to augment and improve the developer's current experience without disrupting the way they work.

3.8 Thoughts about Evidence-Based Programming Language Design

Antti-Juhani Kaijanaho (University of Jyväskylä, FI)

Main reference A.-J. Kaijanaho, "The extent of empirical evidence that could inform evidence-based design of programming languages: a systematic mapping study," Jyväskylä Licentiate Theses in Computing 18, University of Jyväskylä, 2014.
URL http://urn.fi/URN:ISBN:978-951-39-5791-9

In this poster, I presented highlights from my systematic mapping study on the extent of empirical evidence that could inform evidence-based programming language design. I particularly pointed out the curious pattern of publications in this area, where publication seems to have moved from HCI forums to tehcnical programming language forums, and a clear upsurge in publications has occurred in recent years.

In addition, I also discussed my currently unpublished thoughts on empirical evaluation of language design choices, as well as my explication of Evidence-Based Programming Language Design, which will be a significant part of my doctoral dissertation, to be submitted very soon.

3.9 Explaining Software One Bit at a Time

Andrew J. Ko (University of Washington – Seattle, US)

Modern software is increasingly complex, making it ever more difficult to use, understand, and fix. The USE research group invents technologies that help people understand and overcome this complexity, including new help systems for end users, new debugging tools for developers, and new educational technologies for people learning to program. Additionally, we conduct a wide range of studies about software engineering teams, programming expertise, and learning. Our work spans human-computer interaction, software engineering, and computing education.

3.10 User Interfaces for Error Reporting

Shriram Krishnamurthi (Brown University – Providence, US)

Joint work of Marceau, Guillaume; Fisler, Kathryn; Krishnamurthi, Shriram
Main reference G. Marceau, K. Fisler, S. Krishnamurthi, "Measuring the Effectiveness of Error Messages Designed for Novice Programmers," in Proc. of the 42nd Technical Symp. on Computer Science Education (SIGCSE'11), pp. 499-504, ACM, 2011.
URL http://dx.doi.org/10.1145/1953163.1953308

Errors are a critical, inescapable part of programming. How errors are reported therefore has an impact on how programmers perceive the programming experience. This is especially likely to impact beginning programmers, who do not have the experience to know what to do, and are more likely to internalize these mistakes as statements about themselves (with a corresponding impact on their confidence).

Over the past few years, Marceau, Fisler, and I have been studying the quality of error messages in programming environments, and identifying ways to improve them. We have prototyped some new interfaces, which we are now setting up to study.

3.11 Bespoke Tools: Adapted to the Concepts Developers Know

Emerson Murphy-Hill (North Carolina State University – Raleigh, US)

Main reference B. Johnson, R. Pandita, E. Murphy-Hill, S. Heckman, "Bespoke tools: adapted to the concepts developers know," in Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15), pp. 878–881, ACM, 2015.
URL http://dx.doi.org/10.1145/2786805.2803197

Even though different developers have varying levels of expertise, the tools in one developer's integrated development environment (IDE) behave the same as the tools in every other developers' IDE. In this paper, we propose the idea of automatically customizing development tools by modeling what a developer knows about software concepts. We then sketch three such "bespoke" tools and describe how development data can be used to infer what a developer knows about relevant concepts. Finally, we describe our ongoing efforts to make bespoke program analysis tools that customize their notifications to the developer using them

3.12 Bridging between Research and Adoption of Software Tools

Gail C. Murphy (University of British Columbia – Vancouver, CA)

After the hard and creative work of researching, defining and building a tool to aid software development comes the even harder parts: what does it take to get your tool adopted into wide-scale use. Engineering the tool to be easy to use, deployable, scalable, etc. takes some work. But even more work is needed to truly understand the users base, make the tool apparent to use, make the tool fit seamlessly into software development workflows, make the tool handle the multitude of different environments developers use and so on. In the work presented at Dagstuhl, I discussed a number of challenges we faced, and largely overcame, in having a research tool, Mylyn, adopted into wide-scale use and subsequently commercialized in a different form at Tasktop Technologies.

3.13 Using the Natural Programming Approach Throughout the Lifecycle

Brad A. Myers (Carnegie Mellon University, US)

- Make programming easier by making it more natural, by which we mean closer to the way people think about their tasks.
- Apply in all phases of tool development.
- Use a large variety of HCI methods.

3.14 Supporting Developers Decision

Barbara Paech (Universität Heidelberg, DE)

Developers make decisions about the software to be built and the software engineering process. To make adequate decisions about the software they need to understand the users and the former decisions of developers (e.g. about the architecture). The tool UNICASE (www.unicase.org) supports capturing and navigating through the system knowledge (requirements, design, code, test) and project knowledge (e.g. work items) as well as the decisions and their rationale. To understand how developers make and document decisions we have made a study (which is to be submitted shortly) on decision knowledge and decision strategies in issue trackers. This study confirms the importance of naturalistic-decision making. Therefore, in UNICASE decision knowledge can be captured incrementally evolving from a naturalistic decision to a rationale decision with thorough criteria and arguments. In the project URES (http://www.dfg-spp1593.de/index.php?id=38) we study in addition the capture of software usage knowledge. The interaction of the user with the software is monitored on a high-level, e.g. to compare it with use cases in the system knowledge and then improve the the use cases according to actual usage.

3.15 Identifying Barriers to Participation of Females Users on Stack Overflow

Chris Parnin (North Carolina State University – Raleigh, US)

Joint work of Parnin, Chris; Ford, Denae; Smith, Justin; Guo, Philip

It is no secret that females participate in the programming field less than males. This gender gap is also evident on Stack Overflow where in a recent survey only 5.8% of 26,086 respondents identified as female. This study aims to help understand low participation of females on Stack Overflow. Through a manual inspection of users profiles, we found only a 0.25% female participation rate in the top 108,000 user accounts. Through 22 semi-structured interviews with general female users, including an interview with a female user currently ranked as one of the top 100 Stack Overflow users, we identify several barriers, gender-related as well as general barriers specific to Stack Overflow's design. This paper explains why females do not use the site to it's full potential and provides interventions as to how to encourage them to. We found they did not participate, not only because of the time it would take to use the site, but also because they face pressures to research their posts, resulting in them choosing to ask less questions at a lower rate than males.

3.16 The Forgetting Curve

Peter C. Rigby (Concordia University – Montreal, CA)

There has been a great deal of research looking into how quickly students forget what they have been taught. The goal of this research is to investigate how much developers remember of the code they've written. This will help us understand how much knowledge about the system the development team actually has. It will also allow us to assess how long creative professionals retain information about what they've created.

3.17 Does UML Diagram Layout Affect Model Understanding?

Harald Stoerrle (Technical University of Denmark – Lyngby, DK)

Diagrams are widely used in Software Engineering. Intuitively, good layouts are very helpful when understanding UML diagrams. However, existing studies were inconclusive. So, does layout matter?

In a series of experiments we found evidence that layouts do actually matter. We have studied individual factors and the underlying cognitive mechanisms with a view to improving notations and diagramming practice.

3.18 Programming in Natural Language

Walter F. Tichy (KIT – Karlsruher Institut für Technologie, DE)

Main reference M. Landhäuser, T. Hey, W. Tichy, "Deriving Time Lines from Texts," in Proc. of the 3rd Int'l Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE'14), pp. 45–51, ACM, 2014.
URL http://dx.doi.org/10.1145/2593801.2593809

Natural language interfaces are becoming more and more common, because they are powerful and easy to use. Examples of such interfaces are voice controlled navigation devices, Apple's personal assistant Siri, Google Voice Search, and translation services. However, such interfaces are extremely challenging to build, maintain, and port to new domains.

We present an approach for building and porting such interfaces quickly. NLCI is a natural language command interpreter that accepts action commands in English and translates them into executable code. The core component is an ontology that models an API. Once the API is "ontologized", NLCI translates input sentences into sequences of API calls that implement the intended actions. Two radically different APIs were ontologized: openHAB for home automation and Alice for building 3D animations. Construction of the ontology can be automated if the API uses descriptive names for its components. In that case, the language interface can be generated completely automatically.

Recall and precision of NLCI on a benchmark of 50 input scripts are 67% and 78 %, resp. Though not yet acceptable for practical use, the results indicate that the approach is feasible.

NLCI accepts typed input only. Future work will use a speech front-end for spoken input. Better coverage of natural language features is also necessary, for instance for handling repetition and parallelism.

3.19 Industry adoption requires empirical evaluations in a real context

Claes Wohlin (Blekinge Institute of Technology – Karlskrona, SE)

Joint work of Wohlin, Claes and Gorschek, Tony
Main reference T. Gorschek, P. Garre, S. Larsson, C. Wohlin, "A Model for Technology Transfer in Practice," IEEE Software, 23(6):88–95, 2006.
URL http://dx.doi.org/10.1109/MS.2006.147

Research solutions are most often prototypes and not industrialized products, and hence research solutions will not be adopted by industry without evaluation in a real context. This requires researchers to 1) have a strategy for evaluation, 2) have a careful methodological approach, 3) take contextual factors into account, and 4) understand representativeness of subjects.

A seven-step evaluation process is proposed: 1) industry-driven research question, 2) study of state-of-the-art and a joint problem formulation with industry, 3) identify a candidate solution, 4) evaluate the solution in an academic setting (risk minimization), 5) conduct a static evaluation of the solution (dry run, i.e. offline from actual development, e.g. with experts from industry), 6) dynamic evaluation for example a case study in an industrial context, and 7) solution is hopefully ready to be released for industry adoption.

To be successful in this endeavor, context has to be taken carefully into account. This includes 1) understand target context (e.g. business vs. open source) and 2) ensure representative context (e.g. size and type of software system). Furthermore, if assuming use of humans, then subjects in studies are important. This includes two additional factors 3) representativeness of the human subjects, and 4) experience and other relevant attributes of the human subjects.

3.20 Productivity and Data Science for Software Engineering

Thomas Zimmermann (Microsoft Corporation – Redmond, US)

In this poster I present my research related to (1) productivity in software engineering and (2) the role of data scientists in software projects.

Participants

- Andrew Begel
Microsoft Res. – Redmond, US
- Alan Blackwell
University of Cambridge, GB
- Margaret M. Burnett
Oregon State University, US
- Rob DeLine
Microsoft Corporation – Redmond, US
- Yvonne Dittrich
IT Univ. of Copenhagen, DK
- Kathi Fisler
Worcester Polytechnic Inst., US
- Thomas Fritz
Universität Zürich, CH
- Mark Guzdial
Georgia Institute of Technology – Atlanta, US
- Stefan Hanenberg
Universität Duisburg-Essen, DE
- James D. Herbsleb
Carnegie Mellon University, US
- Johannes Hofmeister
Heidelberg, DE
- Reid Holmes
University of Waterloo, CA
- Christopher D. Hundhausen
Washington State University – Pullman, US
- Antti-Juhani Kaijanaho
University of Jyväskylä, FI
- Andrew J. Ko
University of Washington – Seattle, US
- Rainer Koschke
Universität Bremen, DE
- Shriram Krishnamurthi
Brown University – Providence, US
- Gail C. Murphy
University of British Columbia – Vancouver, CA
- Emerson Murphy-Hill
North Carolina State University – Raleigh, US
- Brad A. Myers
Carnegie Mellon University, US
- Barbara Paech
Universität Heidelberg, DE
- Chris Parnin
North Carolina State University – Raleigh, US
- Lutz Prechelt
FU Berlin, DE
- Peter C. Rigby
Concordia Univ. – Montreal, CA
- Martin Robillard
McGill Univ. – Montreal, CA
- Tobias Röhm
TU München, DE
- Dag Sjøberg
University of Oslo, NO
- Andreas Stefik
Univ. of Nevada – Las Vegas, US
- Harald Störrle
Technical University of Denmark – Lyngby, DK
- Walter F. Tichy
KIT – Karlsruher Institut für Technologie, DE
- Claes Wohlin
Blekinge Institute of Technology – Karlskrona, SE
- Thomas Zimmermann
Microsoft Corporation – Redmond, US