CHRISTOPHER J. PARNIN

RESEARCH STATEMENT

RESEARCH INTERESTS AND PHILOSOPHY

My research spans the study of software engineering from empirical, HCI, and cognitive neuroscience perspectives and has been tempered by over a decade of industrial experience. Topics include: workplace interruptions, cognitive frameworks of program comprehension, adoption of language features and tools, software visualization, alternative and live programming environments, usability of developer tools, and crowd systems. Most recently, I explored using fMRI and EMG to actually study the brain activity of developers. I also continue to use large sets of empirical data such as browser histories, instrumented programming environments, and massive online repositories to answer important software development questions (for example, understanding how crowds of developers come together on sites such as Stack Overflow and Github to contribute software knowledge).

My research has been guided by a philosophy: Tackle wide-spread and pervasive problems with solutions framed by theoretical concepts and shaped by real-world applicability. I believe research has the strongest impact when it addresses a problem experienced by many. For example, for my dissertation, I examined workplace interruptions of programmers, an extraordinarily frequent problem with surprisingly little research. By tackling such a wide-spread problem, not only has my resulting research won several best papers and awards, it has become wide-known by practitioners, published in numerous industry trade magazines and featured in industry news and discussion sites.

In the following sections, I describe some ways that the practice of software engineering is likely to change in the coming years, some of the challenges posed by those changes, and some ways that I believe we can address those challenges in research. As the lag between the initial start of software engineering research and its eventual adoption has so far ranged 15-25 years, it is not unreasonable to speculate on the research we need to start now in order to make impact within the next 50 years of software engineering. For an extended version, please feel free to visit the online version

THE NEXT 50 YEARS OF SOFTWARE ENGINEERING

Many of the challenges faced by humanity in the next decades will require software that works at completely different scales and completely different constraints than today’s software. Previously, we’ve been able to make the distinction between programming-in-the-large and programming-in-the-small, when reasoning about the size of teams and types of tools needed to build software. While software continues to fit these situations, it is already diverging from these in the time, distribution, and complexity required. Consider the following scenarios:

- Countries regularly build massive software systems to run government services within a president’s term.
- Software running behind much of the world’s infrastructure celebrate their first century of uptime.
- Billions of disposable apps are created; most are used a few days.
- If today’s developer must know touch interfaces, tomorrow’s must know brain interfaces.

MASSIVELY DISTRIBUTED SOFTWARE ENGINEERING

The development of the Large Hadron Collider’s core software system spanned over two decades, with over 50 million lines of code. Given enough time and dedication, we can create successfully massively large software systems. But, we also may be reaching our limit given our current methods and capabilities. In the United States, the recent software behind the health care insurance marketplace is a reported 500 million lines of code, and is riddled with problems.

As governments increasingly turn legal policy and services into source code and public APIs, often created in the timespan of a president’s term, we must be prepared to build massively-sized software systems on a regular basis. This will often require simultaneous cooperation of many diverse stakeholders. At our current place, imagine how these would fare if needed in a few years: A government api to calculate taxes on all online purchases for any location. A distributed traffic regulation system for a network of self-driving cars and delivery drones.

1http://blog.niniabs.com/2013/12/software-engineering-the-next-50-years/
LONG-TAIL ENGINEERING AND SOFTWARE ARCHAEOLOGY

We must be able to manage day-long and century-long lifetimes of applications. As infrastructure for hosting data and software drops near zero, many instances of software can be created in a few hours, potentially scaling to millions of users, then discarded a few days later. The primary challenge for this type of developer will be not in the creation of the software but in the management of its ecosystem: rapid iteration, instant distribution, insightful feedback.

Like old Roman aqueducts and roads still in use to today, some software essentially becomes eternal, even as the languages, tools, and people behind them are long gone. Other projects—massive in scale—but unable to amass collective resources, must instead plod along over decades. Current research ideas: Reverse-engineering, mining software repositories, program understanding tools may not be enough to ensure the longevity or recovery of knowledge.

NEURAL-EMBODIED AND AUGMENTED PROGRAMMING ENVIRONMENTS

People need to deal with complexity, and software is what we often use to do that. Not only do domains, such as simulating the climate, or understanding flash crashes in automated financial systems, involve immense complexity, software is also increasingly involving a wide range of expertise and skills (end-user programming, cognitive support of quadriplegic and memory impaired). Neuroscience will become essential in understanding how people deal with complexity. When designing software, models of cognitive complexity based on neuroscience principles will be incorporated into software for domains mired in complexity.

Developers will have access to augmented programming environments, which include devices that can sense verbal formations, mental imagery, mental load, alertness, and general brain state. The sensed signals further serve as feedback and input into the programming process as well as other tools, visualizations, and collected meta-data. Furthermore, devices can deliver localized pulses (via transcranial magnetic stimulation) that can prime and enhance particular brain regions needed for programming. Not only will many developers need these devices, they will support other types of knowledge workers using them in their work.

MY FUTURE RESEARCH DIRECTIONS

NEUROSCIENCE OF PROGRAMMING

Performing research on how programmer’s brain’s work has more than just theoretical value but can have real downstream effects in improving education, training, and the design and evaluation of tools, languages, and programming environments for programmers.

Previously, I’ve studied cognitive neuroscience in the context of interruption to programming tasks. In my work, I performed an extensive literature review of cognitive neuroscience of human memory and structured the results in terms of five memory types particularly relevant to programmers: attentive, prospective, associative, episodic, and conceptual. I described how failures in these memory types can be related to empirical evidence of programmers’ information seeking and preservation needs after interruption and designed programming tools that could be used as recovery aids.

- A programmer often takes between 10-15 minutes to start editing code after resuming work from an interruption.
- The worse time to interrupt an programmer is during an edit, especially with concurrent edits in multiple locations.
- Using an episodic review of code changes can often successfully allow a complete recovery from interruption.

Additionally, I have used electromyography (EMG) in studying the different cognitive loads associated with programming. Several colleagues and I have already starting exploring how more techniques such as functional magnetic resonance imaging (fMRI), electroencephalography (EEG), functional near-infrared spectroscopy (fNIR), and other techniques can continue to yield insights into the inner workings of a programmer’s mind.

Finally, numerous DARPA, NSF, and other international grant agencies have been extremely interested in funding interdisciplinary approaches on neuroscience topics.

Questions for Tomorrow

The following list illustrates some the questions we may be able to answer in pursuing this direction of research.

- What does “programmers flow” (a state of complete immersion in a programming task) look like from a neuroscience perspective, and how can we quantify its effects on the programmer and creation of software?
- Programming tools require the concerted use of many cognitive facilities, including symbolic memory, spatial memory, keyboard-based text entry, and visual feedback. How can we design them to take advantage of the cognitive capabilities we are good at and alleviate the challenges of using the ones we are bad at?
- The adoption of end-user software development environments, such as Excel, has expanded the pool of programmers tenfold. What kinds of programming tasks, tools, and contexts will it take to democratize programming to reach the next power of 10? For example, how should initiatives like the “Hour of Code” be designed to account for the cognitive skills and levels of those involved?
- Many philosophies and pedagogies influence how we teach novices to program. How does a teacher’s particular curricular choices affect the cognitive aspects of how students understand, write, and maintain software? How can we use neurological imaging to evaluate what and how well they are learning? For example, does teaching particular concepts such as design patterns change the way people fundamentally understand code?

**Crowd Programming and Crowd Systems**

The software development community has been steadily creating software and tools that allow developers to coordinate on increasingly larger scales. One example of an emergent form of crowd programming comes from the language that everyone loves to hate: Javascript. Rather than having a rich standard API, Javascript essentially has a “crowd API” assembled from Stack Overflow snippets and Github repositories.

Previously, I’ve studied how “crowd documentation”, knowledge created via blog posts and Stack Overflow, for better or worse is increasingly becoming consulted over official documentation and found:
- Developers may be getting as much as 50% of their documentation from Stack Overflow.
- More examples can be found on Stack Overflow than the official documentation guide.
- In web searches, Stack Overflow questions are visited 2x-10x more often than official documentation.

I believe that continuing to study and expand the capabilities of crowd-levels of development not only will better situate us to tackle massive software challenges, but enable long-tail developers to quickly create software for reuse and remix of other developer’s efforts. The techniques used to automatically mine, collate, and extract knowledge from online archives will become invaluable for developers having to maintain century-old software.

This topic has seen tremendous interest in the software industry. I’ve been contacted by several large companies interested in consultation and offering corporate sponsorships on this research topic.

**Questions for Tomorrow**

The following list illustrates some of the questions we may be able to answer in pursuing this direction of research.
- Jonh Carmack has recently quipped that Stack Overflow has saved billions of dollars in programmer productivity. Although many programmers implicitly agreed, we have no current way of measuring so. How can we explain the benefits and costs of arising from crowd systems, where small amounts of software development is distributed across many contributing programmers?
- Despite an accelerating rate of developers contributing to open-source projects and sites such as Stack Overflow, within traditional software organizations, motivations to contribute in public settings do not translate to corporate settings (e.g. resulting in a barren internal wiki). What interactions and incentives will encourage developers to contribute in the workplace?
- Colony collapse: How do you sustain crowd efforts and participation in software? Many, if not most, developers do not yet tweet, blog, contribute to Stack Overflow, or participate on Github. Of the ones that do, just a few perform the majority of work. On Stack Overflow, 60% of questions are answered by 5% of users. Incentives, such as badges, have diminishing effects: Users contribute less as soon as they achieve a badge for the work that earned it.
- How do we automatically mine, aggregate, and curate development knowledge from repositories, sites, and posts? When developers write blog posts about technology topics, they often write in order to reflect their personal experiences or explain how to accomplish a coding task others might be interested in. How might a stakeholder assess and generalize the collective experiences and pain points of its community? How might a stakeholder determine a particular topic is not receiving sufficient coverage?

---