Build it yourself!
Homegrown Tools in a Large Software Company

Edward K. Smith
School of Computer Science
University of Massachusetts
Amherst, Massachusetts
tedks@cs.umass.edu

Christian Bird Thomas Zimmermann
Microsoft Research
Redmond, Washington
{cbird,tzimmer}@microsoft.com

Abstract— **Developers sometimes take the initiative to build tools to solve problems they face. What motivates developers to build these tools? What is the value for a company? Are the tools built useful for anyone besides their creator? We conducted a qualitative study of tool building, adoption, and impact within Microsoft. This paper presents our findings on the extrinsic and intrinsic factors linked to toolbuilding, the value of building tools, and the factors associated with tool spread. We find that the majority of developers build tools. While most tools never spread beyond their creator's team, most have more than one user, and many have more than one collaborator. Organizational cultures that are receptive towards toolbuilding produce more tools, and more collaboration on tools. When nurtured and spread, homegrown tools have the potential to create significant impact on organizations.**

I. Introduction

Tools are a critical aspect of any software development project. They take on many forms. Compiler toolchains, version control, debuggers, test automation, and issue tracking systems are all examples of development tools aimed at making tasks easier. Software development tools come from a variety of sources. Some may be commercial products while others come from open source. The research community is active in creating, evaluating, and disseminating tools to software developers. Tools may be created in-house to help a particular development project as a requirement from project management. However, we have observed that many tools come about not due to management decree, but rather because developers themselves made the decision to build a tool. This may be the result of working conditions to enable a developer to complete his work more quickly or as part of a side project. We term this latter group of tools, developed of developer's own initiative, responding to a local need, and originating in a bottom-up manner through the organizational hierarchy, *homegrown tools*.

Homegrown tools represent a category of work developers undertake that is not mediated by external processes or demands. While homegrown tools might be developed in response to a problem a developer encounters during the source of his or her work, writing a tool is rarely the proscribed avenue of problem-solving. Tools might even be developed clandestinely, without the knowledge or approval of a developer's management. These tools are not tracked in official bug trackers nor stored in official source repositories, making them difficult to study. However, we suspect that informal, homegrown toolbuilding is a common aspect of software development. In our surveys, most developers report building such tools.

Why are homegrown tools worthy of study? Over the past five years that we have conducted research at Microsoft and interacted with development groups in diverse products, we have observed an inordinate amount of in-house tools and have been impressed by the value that these tools bring to developers in the small and entire products in the large.

As just one example of a homegrown tool, CodeFlow [1] is a code review tool at Microsoft that began as a homegrown tool. Clark Roberts and Mike Cook were both developers that felt that the way code reviews were conducted was both tedious and painful. In 2009, they built a prototype of a tool to reduce the overhead of creating and performing code reviews and showed it off in an internal contest for tools and apps within Microsoft's developer division. After being joined by Victor Boctor, a senior architect with similar ideas about how to improve code review, they made CodeFlow available to anyone that wanted to use it. At the time, each team within Microsoft had their own tools and processes for conducting code reviews, but they managed to get their own teams to begin using CodeFlow. Over the next few years, more and more people contributed features and bug fixes to CodeFlow and more teams began using it. Now CodeFlow is the primary code review tool in all product groups at Microsoft. It has been used to conduct nearly five million reviews and currently is used for over 130,000 changes per month by over 30,000 people per month. Over the past year, CodeFlow has transitioned from being a "community" project (a project in which developers volunteer their own time, outside of standard working hours) to a fully funded project in which a small team of developers is paid full time to maintain and improve it. The impact is now extending beyond the company, as many of the design principles in CodeFlow are influencing the code review experience for Visual Studio.

Not all homegrown tools will have the same success and impact that CodeFlow has had. Nonetheless, we have found many examples of others that have become integral to the development projects of the teams that use them. These tools represent countless hours of development, often outside of normal working time. However, they also save development teams' time and increase software quality. Thus, many have a direct relationship with a team's bottom line. It is surprising, therefore, that there has been

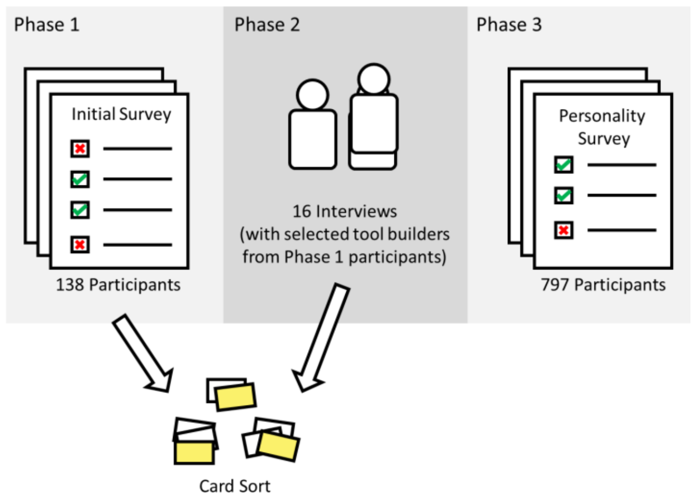

Fig. 1. Experimental methodology diagram. The Phase I survey leads to Phase II interviews. These insights lead to a quantitative personality survey (center left), and open card sorting (center right). These were distilled into a finished research product which you are currently consuming.

little investigation of these tools that result from developers' own volition. In this paper, we mount an investigation into homegrown tools in an effort to provide answers to the following questions:

Who are homegrown toolbuilders?

Understanding the scope, motivations, and extent of toolbuilding begins with understanding toolbuilders themselves. Most initiatives aimed at increasing toolbuilding would be useless if only a subpopulation of developers built tools. Even if toolbuilding is universal, studying the personal factors that contribute to it completes our understanding of the holistic process.

What kinds of tools do homegrown toolbuilders build?

Investigating the types of homegrown tools that toolbuilders build provides insight into gaps in tooling and the challenges that developers face. They may also highlight inefficiencies in current processes and provide avenues for improvement.

Why and when do homegrown toolbuilders build tools?

Exploring the motivations and conditions that lead toolbuilders to build their own tools can enable management and teams to foster environments where developers are more likely to develop beneficial tools. It may also uncover systemic problems that can be addressed (for example, one challenge we observed was tool discoverability within the company).

How do homegrown tools spread?

Understanding how tools spread enables us to maximize spread for most benefit and enhance those channels that work best. Also, understanding how and why tools might *not* spread enables us to understand the reasons behind proliferation of similar tools.

We present a descriptive, exploratory study of homegrown tools at Microsoft. We used surveys and semi-structured interviews to answer basic questions about homegrown tools. In particular, we examined the types of tools that exist; characteristics of the people who build them; what events and conditions cause tools to be built; how tools spread inside organizations; and the impact that tools have.

II. Research Methodology

We collected the data for this study in three phases. In Phase I, we deployed a survey to Microsoft developers to discover baseline information about tools. In Phase II, we followed up with a representative cross-section of toolbuilders and conducted semi-structured interviews in order to find out more about the impact tools have, how tools spread within and between teams, and what attitudes organizations have about tools. In Phase III, we deployed a second survey designed to explore the link between a developer's personality and their tool-building behavior. We present data only from complete survey responses.

A. Phase I: Open-Ended Survey

We conducted a survey of 138 developers and testers at Microsoft to assess baseline levels of toolbuilding. This survey was composed of open-ended questions related to the tools developers had used, grassroots tools they had written, and how they found out about those tools. We used this data to build up our initial ideas about grassroots tools, determine the frequency of toolbuilding, and determine the typical level of sophistication of a tool. The full text of this survey is available for interested readers as a technical report [2] at http://research.microsoft.com/apps/pubs/default.aspx?id=227190.

We sent personalized invitation emails written to 1,000 Microsoft employees with roles related to writing code. We selected the employees at random from the Microsoft organizational database. We have found that personalization and incentives increase participation [3], so we offered participants the option to enter into a raffle for two $50 Amazon.com gift cards. We received 123 responses from our initial population. Since one of the questions in our survey asked if the participant knew any other employees that had written a tool, we sent a second wave of invitations to those engineers and received 15 more responses. Since we intended to use this survey to build a sample for the next phase of our study, this survey was not anonymous.

B. Phase II: Toolbuilder Interviews

After receiving responses from our initial survey, we asked several tool authors from our pool of survey respondents to sit for a semi-structured interview [4] about their tools. Our interview participants included developers, testers, and managers, drawn from the Bing, Office, Windows, and other organizations within Microsoft. We conducted 16 interviews about 12 tools. These interviews ranged between 20-60 minutes in length. We coded transcripts without selecting any *a priori* codes or categories.

C. Phase III: Toolbuilder Personality Survey

In order to further assess the characteristics of toolbuilders responsible for tool building, we decided to conduct a survey of personality data. We decided to conduct this survey because our early interviews yielded less information about the personalities of tool builders than we had hoped, primarily because of participant's hesitancy to talk about more personal topics. Several personality inventories exist in the psychometric research community; the two commonly used in software engineering research are the Meyers-Briggs Type Indicator or MBTI [5], and the Five Factor or "Big Five" model [6]. We selected the Big Five model due to its stronger theoretical and empirical basis, as well as its higher test-retest reliability [7].

*The Five-Factor Model***:** The five-factor personality model refers to five personality domains, called the OCEAN domains by their initials: **o**penness, **c**onscientiousness, **e**xtraversion, **a**greeableness, and **n**euroticism. Over the past few decades, the personality psychology research community has converged on the five-factor model [8] as the standard for assessing human personality traits, and prior research in software engineering that examined personality traits has found success using this model [9] [10]. The five-factor model decomposes personality into five dimensions:

- *Openness* to experience, which measures an individual's creativity, mental flexibility, cultural aptitude, and correlates to intelligence;
- *Conscientiousness*, or will, which measures an individual's will to achieve, responsibility, and follow-through of plans;
- *Extraversion*, the degree to which an individual seeks out social contact;
- *Agreeableness*, the degree to which an individual is friendly and altruistic;
- *Neuroticism*, the degree to which an individual is effected by negative emotional states and moods.

Survey Device: To assess the personality traits of toolbuilders, we used the International Personality Item Pool [11] to construct a 50-item inventory to measure personality according to the Five-Factor model. We sent this survey to 3,000 developers and received 797 responses for a 26% response rate. Since this survey was markedly more personal, this survey was completely anonymous. Participants could choose to email us to enter a drawing for two $50 Amazon.com gift cards. This survey was also longer than the first, containing first the 50-item IPIP personality inventory and later, to justify the effort of the personality inventory, a series of demographic, behavioral, and opinion items totaling 25 questions. This study only reports the findings related to the questions about toolbuilding and their relationship to personality scores and demographics (whether the participant was a developer, tester, or neither at the time, and how long they had been employed at Microsoft).

While the five-factor model performs well on international populations [12], concerns related to the cultural localization of IPIP items led us to distribute this survey only to engineers based in the United States. When piloting the survey, non-native English speakers working outside the United States had trouble understanding the question "How often do you feel blue?" because the term "blue" has different connotations in different cultures, meaning sad in the United States, but intoxicated in some European countries.

D. Data Analysis

This study involved three data sets:

- The survey from Phase I, containing open questions related to tools (abbreviated to *tool survey*)
- The interview transcripts from Phase II. We conducted 16 interviews with homegrown toolbuilders. Table I gives brief introductions to the tools discussed in the interviews.

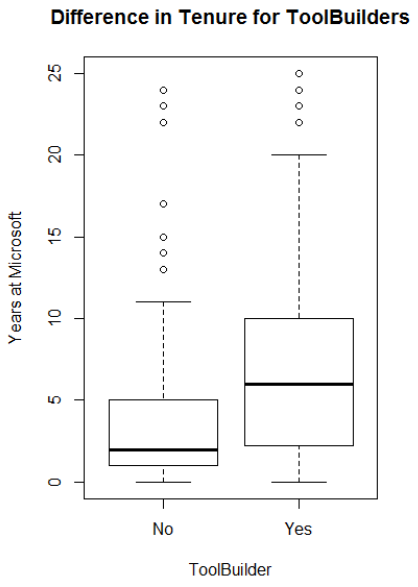

Fig. 2. Boxplots for tenure of non-toolbuilds and toolbuilders

- The personality survey from Phase III, containing closed, multiple-choice questions (abbreviated to *personality survey*)

We analyzed qualitative data using an open card sort [13]. This entailed printing all of our discrete observations on individual cards, then collaboratively clustering the cards into categories. Open card sorts are a natural fit for exploratory studies, because they allow researchers to let a natural organizational system form without pre-existing bias polluting the category structure. We conducted four card sorts, for the following topics: (1) tool types, (2) intrinsic and extrinsic motivations, (3) tool impacts, and (4) tool spread. Our dataset for the card sort included survey responses from the Phase I survey, and transcripts from the Phase II interviews. We generated 564 cards from coded transcripts and surveys that we categorized according to themes that emerged over the course of the card sort. Afterwards, we sorted each category into subcategories.

III. Who builds homegrown tools?

To understand tools, we must consider their builders. In this section, we present the demographic and psychological factors that contribute to toolbuilding. For legal reasons, we were unable to collect data related to gender, ethnicity, or other protected classes, since demographic data for our first study was taken from the Microsoft personnel database.

A. Tenure

Tenure, the length of time an employee has been with the company, is an intrinsic factor in the toolbuilding equation. We computed this statistic from our Phase 3 survey data.

Boxplots with tenure for toolbuilders and non-toolbuilders are shown in Figure 2. Median tenure for toolbuilders in the Phase 3 survey data was 6.0 years, while median tenure for non-toolbuilders was 2.0 years. A Mann-Whitney test of the two groups detected a significant difference in the distribution of the two groups ($p < 0.01$).

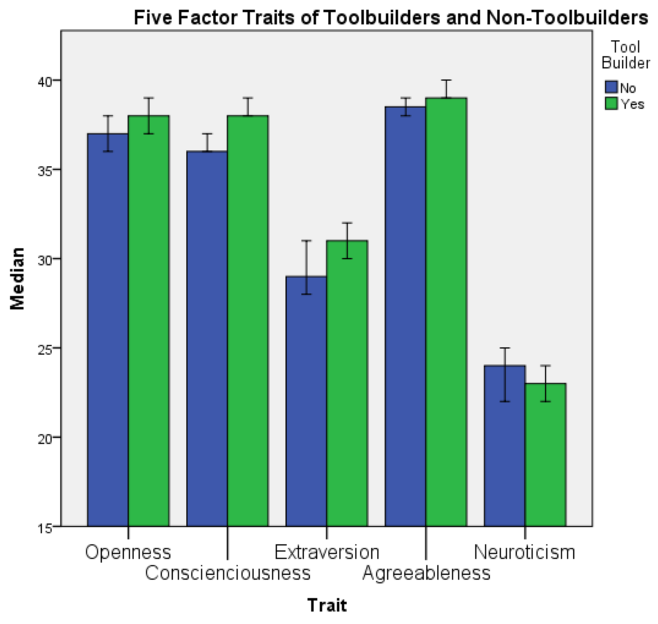

Fig. 3. Median personality factor score for toolbuilders and non-toolbuilders. Scores for a particular factor range from 0 to 50.

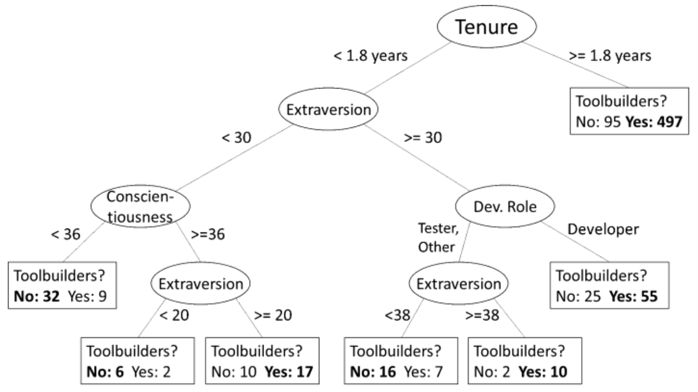
Tenure
< 1.8 years
>= 1.8 years
Extraversion
Toolbuilders? No: 95 Yes: 497
< 30
>= 30
Conscien-tiousness
Dev. Role
< 36
>=36
Tester, Other
Developer
Toolbuilders? No: 32 Yes: 9
Extraversion
Extraversion
Toolbuilders? No: 25 Yes: 55
< 20
>= 20
<38
>=38
Toolbuilders? No: 6 Yes: 2
Toolbuilders? No: 10 Yes: 17
Toolbuilders? No: 16 Yes: 7
Toolbuilders? No: 2 Yes: 10

Fig. 4. Decision tree classifying toolbuilding in our Personality Survey dataset. Numbers below a leaf node are the number of "no" and "yes" cases on the left and right respectively at that node, with the model's prediction in bold.

B. Personality Traits

In total, 597 (74.9%) of the 797 respondents to our personality survey indicated that they had built homegrown tools. We ran an independent-samples Mann-Whitney test on each of the Big Five personality factors across toolbuilders and non-toolbuilders. The Mann-Whitney test detected that toolbuilders are significantly more open (p = 0.041, median difference = 1), conscientious ($p <$ 0.001, median diff. = 2), and extraverted (p = 0.024, median diff. = 2) than non-toolbuilders. Toolbuilders are less neurotic than non-toolbuilders (p = 0.032, median diff = 1). While these differences in OCEAN scores are statistically significant, the small median difference indicates that the effect size is small.

In order to understand and determine key factors related to toolbuilding behavior, we created a pruned decision tree. Fig. 4 presents this tree. The round inner nodes are decision criteria and the edges indicate the criteria used to traverse the tree. Each leaf node corresponds to a group of participants. We label each leaf with the number of participants who do not build tools ("No") and the number who build tools ("Yes"); the majority class is in bold. Decision trees use the most differentiating factor first as decision criteria, in this case the Tenure of an employee.

- Employees who have been at Microsoft for at least 1.8 years are more likely to be toolbuilders: 497 employees have built a tool while 95 have not. No other factor was differentiating for this group in the decision tree.
- For employees who have been at Microsoft for less than 1.8 years (left subtree), differentiating factors between toolbuilders and non-toolbuilders are their personality (the levels of extraversion, conscientiousness) and their role (developers vs. testers and others).

This might suggest that for new employees the personality and development role influences whether they build tools. However, once employees are with the company for a certain period and adapt to corporate culture, personality traits and development role do not differentiate toolbuilder from non-toolbuilders anymore.

The tree is also in agreement with our other findings and intuitions on toolbuilding. We found that toolbuilders have significantly more tenure, and are significantly more extraverted and conscientious. Discussions with employees and anecdotes created an expectation that within Microsoft that testers write more tools than developers (testing requires significant automation), though there was no significant difference in either of our survey datasets (p = 0.775).

IV. What Kinds of Tools do Toolbuilders Build?

One goal of our study was to characterize what types of homegrown tools are being built. Understanding these can give insight into the challenges that developers face and tooling gaps that may exist and may need to be addressed more broadly. The categories that we present come from our card sort of open survey responses gathered in Phase I and the interviews conducted in Phase II.

A. Common types of tools

Here we present an overview of the descriptions of tools that developers built. Each category emerged during our card-sorting process as described in Section II.D. These categories represent qualitative distinctions between tools emerging from our discussions, not an attempt at an objective taxonomy of tools.

Information Retrieval – Information retrieval tools access and report specific information to their users. Information retrieval tools locate, process, and display information on-demand for users.

Testing – This category represents any tool related to testing software. This may category includes tools such as test automation, test reporting, or tools that actually conduct the testing. We describe an example of such a testing tool called xAuto in more detail in the Section B.

General Automation – The general automation category represents any tool unrelated to testing, building, or deploying specifically that automated a previously manual process.

Debugging – We categorized any tool related to tracking down a specific defect in software as a debugging tool. MemSpect (a tool we will describe in Section D) is an example of a debugging tool.

User Interface – User interface tools provide ways to use existing functionality in a different, usually graphical, interface. Microsoft contains a diversity of services and processes that might lack a visualization or graphical interface, e.g., because it runs on the command line, prompting developers to build their own UI.

Deployment Automation – Deployment automation tools relate to installing software, usually for the purposes of testing new builds. While this is a common task, complex build processes and runtime dependencies as well as diverse environments can make deployment difficult, necessitating automated tools.

Monitoring – We categorized any tool that persistently watched for a set of events as a monitoring tool. Many monitoring tools were information retrieval tools that ran constantly, or tools that automatically performed an action given some trigger, such as a build being completed or source code being checked in.

Extensions – Many commercial tools provide extension points or plugin facilities. This category comprises all the quotations we found that described extensions of existing products or tools, and which did not belong in a more specific category. One notable case is the IDE. Microsoft developers spend most of their time in the Visual Studio IDE, and some have used Visual Studio's extension framework to add functionality. DiffButler, discussed later in this section, is an example of an IDE-based tool. One of our interview tools, SuiteNinja, is a Visual Studio plugin intended to be a general grab bag of common actions performed by a specific team.

Build-Related Tools – Many developers referred to the build process that they deal with. The tools described range from simple build automation systems to more sophisticated actions that ran as part of a build. One developer mentioned a tool that added additional information to a build; another tool extracts information from a new build.

We received a small number of responses indicating other types of tools, which we categorized into: Personal Support Tools, which facilitate communication and information management with team members; Machine Learning Tools, which relate to building machine learning models; and software libraries. Notably, VSO Cortana is an example of a personal support tool.

In the following subsections, we present three vignettes that illustrate toolbuilding scenarios we encountered, to provide a more comprehensive picture of what factors drive tool creation, the challenges and goals in toolbuilding, how teams react to homegrown tools, and how they spread. While these vignettes are not intended to be representative in a statistical sense, they illustrate diverse points throughout the space of homegrown tools. All names are anonymized.

TABLE I. Tools discussed in interviews

Tool Name	Description
MemSpect	A memory debugger for hybrid managed/native applications.
DiffButler	A visual studio extension that displays an inline diff of the file the user is editing.
Agile Support Tools	Assorted tools to support an agile team workflow.
Watchdog Video Viewer	A WTT log viewer for Windows interface tests that plays screen captures synchronized to the test log.
Suite Ninja	A Visual Studio extension that automates common actions for its home team.
MSMQ Viewer	A utility to view and edit Microsoft Message Queuing queues on Windows servers.
xAuto	A distributed, randomized "chaos monkey" testing system.
Build Status Monitor	A tool that monitors for new Windows builds and automatically copies the build to test machines and extracts build metadata.
Damascus	A webservice that continuously checks development environment setup scripts and reports when an error occurs.
VSO Cortana	A Cortana-based app for Windows Phone that allows developers to close tasks, view build status, and perform other actions in Visual Studio Online.
CrunchNet	An automated diagnostic tool for network captures.
Test Pass Manager	A monitoring service for test runs.

B. xAuto

When the OneNote team was first building collaboration features, they encountered a data corruption bug that only manifested within the first day after interacting with a shared notebook. Typically, internal beta usage data would isolate the problem, but it was difficult to find enough users of the beta software on this new feature in a still-nascent product. Adam, a test lead, wrote a random testing tool that he called OneAuto to in an effort to find the bug. The bug was so critical that the entire team ran instances of OneAuto on their machines overnight. OneAuto randomly selects edits to apply to a shared notebook and then randomly passes control to another instance on another machine, which then does the same thing.

Before it had found the bug it was created to track down, OneAuto proved its value by discovering other bugs. Most importantly, the OneNote team was able to fix these bugs, because OneAuto was discovering problems with a high degree of accuracy. Even today, while directed automated scenario testing finds more bugs in total than xAuto, the bugs xAuto discovers have a 40% fix rate, which interviewees indicated is considered high for an automated tool. Further, because OneAuto worked through the existing extensibility system, and since testers were obligated to test the extension points for their features, OneAuto quickly grew to support all OneNote features. Impressed by the tool, Adam's manager, Barbara, evangelized the tool heavily to other groups within Office.

Soon after the team fixed the collaboration bug, Adam left the group to head another team. OneAuto development responsibilities fell to Claire, a developer in test who was an intern when OneAuto was first developed. As more products added collaboration features and Barbara continued evangelizing the tool, OneAuto attracted mounting attention. At the same time, more of the products within Office began implementing collaborative co-authoring. When Word implemented this feature, they forked OneAuto to help with testing, having heard of it from Barbara. To avoid the inefficiency of a fork, Claire worked six months full-time on OneAuto, collaborating with David, another developer in test in OneNote. When she was finished, OneAuto had become fully generic, and its name changed to xAuto. The Word team quickly adopted xAuto, and the tool began to spread within Office.

Later, Barbara moved from OneNote to the Project group inside office and was replaced by Edward. Edward had formerly worked with Adam and had previous experience with genetic-algorithm driven smart monkey testing. He continued to champion the project as it spread within Office. Eventually, Claire became a test lead, and switched from developing xAuto to managing it. A "virtual" team built of developers and testers from across Office, headed by David, currently maintains the tool. As of today, every team inside Office has added support for their product to xAuto, and the tool has become ubiquitous within Office.

C. DiffButler

One popular homegrown tool within Microsoft is Odd, a diff viewer. Frank was a heavy user of Odd, because he likes to have a reference of what changes he's made in a code file. However, after becoming annoyed at how often he had to switch windows between Odd and Visual Studio, Frank decided to write a tool.

DiffButler is a Visual Studio add-on that highlights lines and tokens that a developer changes. If a file is tracked in source control, DiffButler will use the last version checked into source control as the base file rather than the file as it exists on disk. While Frank has told his immediate team about DiffButler, he thinks no more than one or two of them use it. Frank said that while some developers might find inline diffing valuable, others might not need it.

DiffButler is shared on a Microsoft-internal site similar to SourceForge or GitHub that hosts downloads, issue tracking, and source control for homegrown tools inside Microsoft. Frank usually updates DiffButler to work with new versions of Team Foundation Server or Visual Studio once every few months to annually. The most recent version has more than 300 downloads.

D. MemSpect

About three years ago, Grace was assigned a memory leak bug in Visual Studio – when a certain feature was exercised in a loop for 20 hours, a memory leak occurred which eventually starved the host machine of memory. Grace realized that the only way to track down the bug was to write a tool that could inspect Visual Studio's memory while it was running. Grace searched for existing tools, but only found tools that worked on fully native code. At the time, Visual Studio had begun to transition to being a hybrid managed and native application, preventing the existing tools from being useful. To track down and fix the bug, Grace wrote MemSpect.

Gradually, Grace gained a reputation as "the memory guru", and as her colleagues came to her with memory issues, she taught them to use MemSpect. MemSpect gradually spread further as Grace presented it in various contexts. MemSpect even won awards inside Microsoft.

MemSpect is developed entirely in Grace's personal time; although it now has hundreds of users by her estimation, it does not contribute directly to Grace's team's bottom line, and so she does not receive time allocation for it. While she has received many requests to add support for 64-bit applications to MemSpect, it is unlikely to be implemented because Visual Studio is 32-bit.

V. When and Why do toolbuilders build tools?

We have described who builds tools and the types of tools they build. We also investigated the conditions, needs, and desires that led to building a homegrown tool.

By our definition, developers do not build homegrown tools because that are told to. Rather they make the choice themselves as a result of internal and external factors. We posit that at some point, developers decide that the cost of building a tool is outweighed by the cost of continuing without it. At this "creation moment" a developer begins building a homegrown tool. We describe each in more detail here. Note that often, more than one of these factors may come into play when deciding to build a homegrown tool.

Save Time – The most common reason for developers to build tools is to reduce the time a process takes to execute. This could be something that they would otherwise have to do manually or something that takes time on their team. For example, Test Pass Monitor is a tool that checks if Windows tests runs have completed or have stalled; previously this check was performed manually, and the responsibility for performing it cycled through the team. By writing a tool that does this, the team was able to eliminate the distraction of micromanaging test runs and put that time back into their cycles. The MSMQ viewer developed by the Office Engineering team allowed the team to spend orders of magnitude less time debugging errors involving the Microsoft Message Queuing system in Windows Server, by enabling them to quickly see an overview of the contents of a queue and edit messages dynamically.

Help Others – People are empathetic by nature; sometimes they express empathy through altruistic toolbuilding. Some developers expressed their wishes to make other people's lives easier, and built tools to address pain points for team members.

Reduce Pain – We posit that developers might have a lower pain threshold for automatable activities than other demographics, due to their ability to automate many of their daily tasks. This ability to automate can make manual tasks more annoying or mentally painful, leading developers to automate away these annoying tasks. This was particularly the case for the tool Build Status Monitor, which exists primarily so that its developer could stop having to manually find build identifiers present in a build's directory structure, a process the developer found irritating.

Personal Need – We coded responses as reflecting a personal need if they contained a statement about the developer's own needs separate from their business needs or team needs. Personal differences between a developer and their environment might lead them to write tools that address those specific differences. DiffButler is an example of a tool motivated by a personal need.

No Known Solution – The most common reason people report building tools is because they are not aware of an existing tool that does what they want. In some cases, such tools may exist, but while Microsoft has internal sites for sharing tools, not all developers share their tools on them. When they do, their tools still might not be discoverable enough to prevent all duplicated work.

Different Environment – When a tool does exist that matches most of what a developer wants it may not work in that developer's environment. For example, while memory debuggers existed for native code, MemSpect's author was unable to find any that would work for Visual Studio, which was moving to managed code at the time MemSpect was designed. Since Microsoft is so vast, many teams have different or dedicated infrastructures that prevent tools they build from being useful in other environments. As powerful as xAuto is, it remains tied to Office infrastructure that prevents it from being used by Visual Studio or Windows developers. The most common cause of environmental differences is change. As languages, frameworks, and feature sets change, the existing tool ecosystem built around them are rendered obsolete. The migration from C++ to C# had a dramatic effect on tooling; the transition from waterfall to agile development at Microsoft was mentioned as a factor for building Agile Collaboration tools, VSO Cortana, and another tool mentioned during the Damascus interview.

Management Support – Possibly the most vital factor for toolbuilding is a supportive management. Homegrown tools are, by definition, outside the scope of an engineer's normal work, so if an engineer's management is not supportive of toolbuilding, the engineer can use only their free time to build tools. However, some managers are generally supportive towards toolbuilding. We observed this attitude in both the test and development disciplines, but support for toolbuilding was more consistently present in the test discipline, possibly because automated testing can have a more relevant and immediate return on investment than development support tools. For example, when the author of Test Pass Monitor spoke about his manager's reactions to his tool, he described the manager as being *"really happy about it actually."* Suite Ninja was developed as part of a push from the VP-level to improve tool support in its org. The Suite Ninja interview participant said that *"our management is very supportive of anything that can free up our time, or is beneficial to the team overall – you do the work once and everybody benefits, and generally management's very much for that."* Some managers see toolbuilding as a long-term investment – as Barbara, the test manager from our xAuto vignette said, *"It can sometimes take years [for a tool effort I support to be finished]. I'll be patient."*

Management Barriers – Management culture can also be hostile to toolbuilding. Several of our interview participants thought their managers would not support their work on tools because their primary responsibility is to implement features. This might be a systemic difference between development and test roles: virtually all of the environments we observed that were not supportive of tools were in the development role. Conversely, testers can work on tools that directly relate to their jobs in a more meaningful way, and an effective test tool can serve as a productivity boost for a test team. The productivity gains for a tool that a developer could use are more indirect, and harder to measure.

While Microsoft is trying to become more supportive of internal innovation, other policy changes might work against this goal. For example, one interview participant noted that in the older waterfall development cycles, team members could repurpose slack space between large development efforts to build tools. In the newer Agile development practices, these slack times no longer exist, leading to fewer opportunities where developers can reinforce their tooling.

Business Need – We coded responses as a business need motivation if the quotation mentioned building tools to make Microsoft better as a company, or to respond to an immediate business need. For example, one participant identified an internal infrastructure-consolidation effort as a motivation for building a tool.

Reduce Error – Homegrown tools sometimes emerge to reduce the amount of error in a pre-existing process. Suite Ninja is a Visual Studio plugin that allows developers to force an association of a monitoring ID with their code branch. Previously, this process occurred days or weeks later as part of a build step. Delaying the association meant that conflicts sometimes occurred, causing alerts from faulty branches to go to the wrong developers. Suite Ninja was motivated in part by desire to prevent these errors.

Assignment – While our definition of homegrown tools preclude tools created by assignment, tools that rise organically might be adopted by their organization. As grassroots tools mature and prove their value to a team, the team might allocate official resources towards that tool. This was the case for three of our interview tools: Agile Support Tools, MSMQ Viewer, and xAuto. One survey respondent even said, *"I was tasked to work on this tool as a project"* when asked why they began working on extending a homegrown tool.

Centralize Expertise – Some developers built tools in an effort to consolidate expertise in a process. A developer might build a tool to enable other people to perform actions that otherwise require significant domain expertise. The tool CrunchNet is an example of this; it allows software engineers without significant networking experience to determine if the network is causing a failure in a deployed service. Without the tool, this would require manually inspecting a packet trace.

Team Culture – Some teams have a shared culture of building and distributing tools among themselves. The Damascus authors described a typical interaction: *"We just talk about [tools we build]. I say, 'Hey, remember that problem we ran into last night? I wrote this thing last night, you should use it now.'"* When asked, several interviewees said that their team had a tradition of building tools collaboratively, though none of them felt that was typical.

VI. How do homegrown tools spread?

Many of the tools that we encountered spread beyond their original developers or teams. It is difficult to accurately determine how a tool spreads and assess its impact because homegrown toolbuilders may not be aware of who is using their tools or how others become aware of them; this prevents us from presenting authoritative, quantitative data on tool use and spread. In addition, we found that many toolbuilders did not have grand aspirations for their creations and instead were happy to have their tools used solely by their teams or just themselves. Some expressed the notion that the more people using a tool means more bugs to fix and feature requests, something developers avoid in their normal routines, let alone their spare time.

Nonetheless, effective tools can have a larger benefit if more people are using them. Tool spread can also avoid duplicate work in the form of developers writing their own very similar tools. We therefore investigated the various channels by which tools spread and the ways that they have had impact in terms of users and time savings.

Cross-Product Collaboration – Within Microsoft, different products have a significant distance between them, and are often developed in entirely different infrastructures. However, some grassroots tools displayed collaboration across products. Most of these quotations originate from interviews related to xAuto, but DiffButler, MSMQ Viewer, CrunchNet, and MemSpect have also crossed product boundaries.

Sanctioned Channels – Microsoft contains a number of official channels where developers are encouraged to share side projects. These include:

- The Garage, a Microsoft-wide hackerspace that offers trainings, hosts talks on new technologies, and has weekly hack nights and demos where Microsoft employees and interns can share side projects
- Organizational hackfests, events where all teams in a particular organization create, pitch, and execute ideas
- Organizational "science fairs" or other presentation days, where engineers can register a booth or talk about their side project and demonstrate it to their peers and managers
- Less formal presentations and brown bags (informal lunch presentations) within teams or groups of closely-linked teams

Many of the tools from our sample were influenced by these sanctioned toolbuilding channels. The developers of VSO Cortana were encouraged by the Garage community to develop their app, and got the idea to make it cross-platform after a demonstration on Xamarin hosted by the Garage. We met the VSO Cortana developers at a Bing Science Fair, an annual event where developers are encouraged to present their tools to their peers and compete to receive recognition by a panel of expert judges. The authors met the developers of Damascus and CrunchNet at the same science fair. Damascus developers credit the science fair with giving them motivation to finish Damascus: *"We had the code, and then we saw the science fair stuff, and thought maybe we should finish it for that, so again we talked to our manager and said we need to be able to spend time on this."* Suite Ninja was proposed and developed during a similar Hack Day.

Collaboration – Collaboration was typical in the tools studied. Since homegrown tools aren't subject to the security constraints surrounding product code, it is easier for developers to collaborate on them. For example, xAuto's current maintainer David described it in our first interview as *"essentially an open source tool within the Office community."* Of the other tools whose creators we interviewed, the MSMQ Viewer, Watchdog Video Viewer, MemSpect, and Suite Ninja were developed collaboratively.

Direct Contact – One of the most common ways tools spread is by their developers telling other people about them. The developer of Watchdog Video Viewer said that his users found out about his tool by, *"basically my telling them."* When the developer of Test Pass Monitor started using it to replace his own shifts manually watching tests, he configured the tool to send an automatic email with a link to the tool's internal project page to the testers who owned the current test run. To some extent, all of the developers we interviewed had directly spread their tool to new users.

Hierarchical Spread – Many tools in our dataset spread hierarchically throughout Microsoft – spreading up the corporate hierarchy from a developer to a manager, and then out to the rest of that manager's reports and peers. This is most common in environments that are more receptive to toolbuilding, xAuto notably had a number of management evangelists early in its lifespan who assisted its spread from the OneNote team to others inside the Office organization. In another instance of hierarchical spread, the developer of Test Pass Monitor related how his tool came to be used by the rest of his organization: *"My manager's manager found out about it, and scheduled a meeting with all the other leads in my group. I gave a quick presentation on how the tool works, and from there it spread."*

Team Use – The first people a developer tends to share a new tool with are those nearby – namely, their team. Each of our interview tools had users on the developer's team, with the exception of VSO Cortana and Damascus, which are currently unreleased.

Low Barrier to Entry – Some of our quotations explicitly discussed the low barriers to entry that their tool exploited, leading to more rapid adoption of the tool. xAuto is a particularly good example of this, having hit several sweet spots early on. As one xAuto developer put it, *"Since [xAuto] was built on a shipping extensibility model, [and] each one of the testers had to test extensibility for their feature in general, they had the knowledge of how to write the ability to add a page, or whatever features they had… they just needed to plug it into this framework, and I tried to make that as straightforward as possible."* Additionally, xAuto required no installation – *"you just had to have the exe on the machine."* Later, as xAuto was spreading through Office, it benefited from having a reserved lab for xAuto runs – meaning that new teams didn't have to allocate computing resources to run xAuto. Without taking advantage of these previously existing systems to create low barriers to entry, xAuto and other tools might not have spread as far.

Uncertain Spread – Many developers we talked to were uncertain of the extent to which their tools had spread. This was usually the case for personal tools that the authors had shared on internal sites, since they had no way of knowing the active users of their tool beyond how many times a particular version has been downloaded. The developer of the Build Status Monitor tool said, *"I'm not sure what the usage of the tool is,"* and the developer of Watchdog Video Viewer similarly said, *"I don't know if anyone else has used it on a regular basis. I know about 56 people have downloaded the tool."*

Needs Differences – Differences in the needs of user groups affects the spread of tools. As the author of DiffButler said in an interview, *"I still have the hunch that there are varying types of developers out there, and some don't need this and some do."* While individual differences might prevent someone from using a personal tool, organizational resources and needs might preclude. For example, differences in the overall quality goals between Bing and Office mean that Bing is very unlikely to use xAuto.

Social Spread – Some tools spread via a social network. In a large company like Microsoft, many developers have previously been on different teams with different people. When developers move teams, they might bring new tools with them, or bring up a tool they've heard of. When xAuto was first spreading to other teams outside OneNote, the first teams to adopt it were teams containing developers that had previously been in OneNote or on other teams with people related to xAuto. Another tool discussed with the MSMQ Viewer developer spread outside his team for the first time when a former team member, having just moved to a new team, was appalled at the error-prone nature of their database update procedure and helped them adopt the tool his previous team used to orchestrate the process.

Barriers – Some of the developers we talked to had chosen not to share their tools. For personal tools, a big obstacle was legal liability, and uncertainty about the policies regarding sharing within Microsoft. For example, the developer of DiffButler remarked he would like to share his tool externally, but *"wouldn't even know where to start with something like that."* Similarly, an xAuto developer didn't spread the tool early on because he "didn't know [he] had the scope to actually go out" and spread the tool.

A team that supports a tool might not share it because that would require some amount of additional work the team cannot justify to themselves, since the tool is already usable to them. A developer of xAuto said, *"I don't think we've published it further out because we didn't know how to maintain it."* Teams might also not want to publish work that reflects badly on them – the developer of the MSMQ Viewer said, *"We would want to clean it up a little bit more in terms of ...the UI."* Homegrown tools are typically side projects, so polishing them to the point at which other teams can use them might not be a priority.

VII. Implications

What are the implications of this research? While we have gained an understanding of homegrown toolbuilding, there are also additional lessons learned that can benefit software development organizations. Some of these recommendations come from things we observed already in practice at Microsoft. Others come from shortcomings that can be addressed. The value and implementation of each is dependent on the context of the particular development team or organization, as startups and open source projects may face different challenges or have different needs.

Hackdays - Organizations should embrace and encourage toolbuilding through hackdays. One of the most valuable practices we observed in our study was organizational hackdays. These hackdays consumed between two to four days out of a year, typically, but produced a multitude of valuable tools and provide engineers with an opportunity to exercise their passion for toolbuilding. Organizational hackdays and tool presentation days allow for toolbuilders to be recognized for their hard work, and for valuable tools to spread within an organization. We observed that engineers feel uncertainty about the appropriateness of spreading tools – hackdays alleviate this uncertainty by providing a defined structure for developing and spreading tools. Hackdays are a cheap investment with potentially massive returns.

Discoverability and Evangelism - A common reason that developers built tools was that they were unaware of existing tools that matched or could be easily adapted to their needs. In some cases, such tools do exist, but they may be difficult to find. Microsoft has an internal site meant to host community projects that alleviates some of the problem. Teams and individuals should be encouraged to use such resources even if they question that others would find their tools useful. Further, curation and categorization would make more useful and appropriate tools easier to find.

Currently, Microsoft provides opportunities for developers to present their tools to other teams (e.g., informal lunch talks). In addition to this, individuals and teams should have venues to discuss their problems and challenges with other teams that can lead to collaboratively using or improve existing homegrown tools rather than reinventing them.

Plan Transitions - One common reason for building a new tool is that previous solutions exist, but only for a *different environment*. We noticed that these environmental differences frequently come about because of a longer-term transition. MemSpect was hardly the first memory debugger, but it was the first (within Microsoft) that could handle managed programs. xAuto is the latest in a long line of monkey testing systems, but was the first to test collaborative co-authoring. VSO Cortana and Agile Support Tools, as well as others discussed in interviews but not named here, came about because of a tool gap exposed after the transition from waterfall to agile development within Microsoft.

Because of this, tooling should be taken into account when technology or process transitions are considered. Organizations should plan for adapting their tools and consider the cost of losing tools that are not adaptable. Where large teams or organizations make a transition, they should take care to make sure that there isn't redundant work going on in their teams developing similar tools to deal with the new environment or processes.

Developers should also consider how much a tool needs to be tied to a particular environment or system. If relatively painless design decisions up front can allow tools to be adaptive later, they can yield a savings of effort. CodeFlow was designed from the start to be loosely coupled with the repository system. As a result, intrusive changes were not required when adding git support.

Tool Culture - Teams and organizations should encourage *tool culture*. During this study, we often found ourselves discussing a holistic concept of *tool culture.* Organizations with *tool culture* are friendlier towards experimental making and toolbuilding. Beyond management practices, tool culture extends into the attitudes team members have towards solving problems and collaborating. A team with tool culture is likely to have several tools in various stages of completion kicking around on developer's computers. While almost none of these tools will be valuable on the level of the organization, this practice has two key valuable aspects. The first is the attitude towards solving problems via toolbuilding; the second is team collaboration on promising tools.

Broken Windows - Another interesting anecdote related to us in our interview of the Damascus developers was the concept of "broken window effects." Taken from the sociological-criminological broken window theory [14], a "broken window" as applied to software engineering is an element that is visible, and broken, such as inaccurate documentation or broken setup scripts. According to broken window theory, broken windows beget broken windows. A developer working on a project with an outdated wiki, frequently incorrect documentation, and some persistent bugs might not put as much effort into unit testing, writing performant code, or keeping interfaces "clean." Homegrown tools can play a powerful role in preventing this effect, since the project components most likely to become "broken windows" are those that are considered ancillary rather than central, much like homegrown tools. Teams that are friendlier to homegrown tools, valuing investment into team infrastructure more than strict adherence to assigned work, might stave off broken window effects more readily than teams hostile to homegrown tools.

VIII. Validity

This work is a qualitative case study; we do not attempt to broadly generalize from our conclusions here. Since our surveys and interviews were conducted with only Microsoft engineers, we cannot make any overbroad conclusions. However, Microsoft is a large company with a great degree of internal diversity with respect to software engineering practices, and its employees come from a wide array of backgrounds. That being said, our findings are likely not reflective of typical open source projects where participants do not work together in a single company and may not have the same impetus to create or evangelize homegrown tools.

While our first survey was advertised as a tool survey and therefore could have been subject to self-selection bias, the second survey was not advertised as a tool survey, and should have no self-selection effect between toolbuilders and non-toolbuilders.

IX. Related Work

Homegrown tools are a specific case of the general concept of grassroots innovation, management, and cultivation. This business strategy, while not extensively studied in computer science, is more commonly studied in management science. In computer science, Bailey and Horvitz examined the grassroots innovation in Microsoft, outlining the infrastructure that exists to cultivate innovations in various business areas [15]. While the system they describe was not used for building tools during the course of their case study, it is similar to the organizational hackfests we observed. The management and organizational science community contains more research on grassroots innovation. Brand presents a review of 3M's practices for fostering grassroots innovation, notably it's "15% rule", the earliest example of allocating time for employees to pursue their own projects [16]. This practice was later adopted by Google (as "20% time") and HP. While some results of these innovation processes are public, such as the invention of the Post-It note at 3M and of Gmail and Google Earth at Google, not all inventions occurring as a result of these policies are publicized and a selection bias exists in that failed grassroots innovations will usually never be made public. We are unaware of any work on the types of tools produced by these innovation pipelines. More broadly, Andriopoulos presents a review of the management science literature pertaining to creativity within organizations that includes the freedom to experiment and self-directed activity as factors contributing to creativity [17]. Outside of industry, Bardzell et al documents the toolbuilding behavior of individuals in hackerspaces [18].

We used the OCEAN or Five-Factor/Big-5 personality metric to attempt to answer the question of who builds homegrown tools. Other examinations of the effect of personality on development behavior include Salleh et al's and Hannay et al's work on the effect of personality on pair programming behavior [9] [10] [19], and Licorish and MacDonell's work on inferring personality types for distributed developers using artifacts and using the data to explain how personality effects general development [20]. Personality is widely studied with respect to pair programming, specifically the problem of choosing which developers to pair. The Meyers-Briggs inventory has also been used to examine personality factors in software engineering [21] [22].

Tool adoption and tool spread within organizations and communities is a well-studied problem in software engineering and other research communities. Diffusion of Innovation theory is a popular framework for studying the transmission of ideas through communities [23]; Xiao et al [24] applies this framework to security tools. Within computer science more broadly, the Socio-PLT project has investigated the factors related to spread and adoption of programming languages [25] [26].

The impact of specific tools on development processes has not been widely studied. The closest similar research topic is cost modeling and prediction, the problem of predicting how costly a particular software project will be to implement. Some models, including popular COCOMO family [27], include the use of software tools as a component of cost, but only examine the raw availability of tooling rather than tool quality or development cost in itself.

X. Conclusion

The goal of this work was to explore the space of homegrown developer tools, in order to better understand what motivates toolbuilders and what effects their tools have on their teams and organizations. To this end, we conducted two surveys and a semi-structured interview campaign. In our samples, most developers reported building tools. We found that toolbuilders had statistically significant differences in the Openness, Extraversion, Conscientiousness, and Neuroticism scores on the Five-Factor personality model, as well as being significantly more tenured. We show that homegrown developer tools are diverse, ranging from test systems to IDE plugins to mobile applications, and that they are born from a wide array of circumstances and team cultures.

Many of the tools we observed were born of necessity, and many have high impact relative to their development cost. The implication we take from this is that organizations should encourage a healthy culture towards tool building through organizational hackfests, and make their tools discoverable in order to maximize the benefit they can gain from homegrown tools.

We hope this work will be the first of many academic inquiries into tools developers take the initiative, the time, and in some cases, the risk, to build themselves. An understanding of the tools developers find important to build, and the tools that provide organizational benefit, would be of great value to the software engineering research community, focused in large part as it is on building new tools that enable developers to work more efficiently and produce better software. This understanding could deeply strengthen the relationship between the producers of research and its intended consumers.

XI. References

[1] Microsoft Corp., *Codeflow,* http://www.microsoft.com/en-us/news/features/2012/jan12/01-05codeflow.aspx, 2012.

[2] C. Bird, T. Zimmermann and E. K. Smith, *Appendix to Do It Yourself! Homegrown Tools in a Large Software Company,* 2014.

[3] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird and T. Zimmermann, "Improving Developer Participation Rates in Surveys," in *Cooperative and Human Aspects of Software Engineering*, 2013.

[4] T. R. Lindlof and B. C. Taylor, Qualitative communication research methods, Sage, 2010.

[5] I. Myers and P. B. Myers, Gifts differing: Understanding personality type, Consulting Psychologist's Press, 1980.

[6] P. T. Costa and R. R. McCrae, Revised neo personality inventory (neo pi-r) and neo five-factor inventory (neo-ffi), vol. 101, Psychological Assessment Resources Odessa, FL, 1992.

[7] D. J. Pittenger, "Measuring the MBTI… and coming up short," *Journal of Career Planning and Employment,* vol. 54, pp. 48-52, 1993.

[8] J. M. Digman, "Personality structure: Emergence of the five-factor model," *Annual review of psychology,* vol. 41, no. 1, pp. 417-440, 1990.

[9] N. Salleh, E. Mendes, J. Grundy and G. S. J. Burch, "An empirical study of the effects of conscientiousness in pair programming using the five-factor personality model," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010.

[10] N. Salleh, E. Mendes, J. Grundy and G. S. J. Burch, "An empirical study of the effects of personality in pair programming using the five-factor model," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009.

[11] L. R. Goldberg, J. A. Johnson, H. W. Eber, R. Hogan, M. C. Ashton, C. R. Cloninger and H. C. Gough, "The International Personality Item Pool and the future of public-domain personality measures," *Journal of Research in Personality,* vol. 40, pp. 84-96, 2006.

[12] R. R. McCrae and P. T. Jr, "Personality trait structure as a human universal.," *American psychologist,* vol. 52, p. 509, 1997.

[13] D. Spencer, Card Sorting: Designing Usable Categories, Rosenfeld Media, 2009.

[14] J. Q. Wilson and G. L. Kelling, "Broken windows," *Atlantic monthly,* vol. 249, pp. 29-38, 1982.

[15] B. P. Bailey and E. Horvitz, "What's your idea?: a case study of a grassroots innovation pipeline within a large software company," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010.

[16] A. Brand, "Knowledge management and innovation at 3M," *Journal of knowledge management,* vol. 2, pp. 17-22, 1998.

[17] C. Andriopoulos, "Determinants of organisational creativity: a literature review," *Management decision,* vol. 39, pp. 834-841, 2001.

[18] J. Bardzell, S. Bardzell and A. Toombs, "now that's definitely a proper hack: self-made tools in hackerspaces," in *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, 2014.

[19] J. E. Hannay, E. Arisholm, H. Engvik and D. I. Sjoberg, "Effects of personality on pair programming," *Software Engineering, IEEE Transactions on,* vol. 36, pp. 61-80, 2010.

[20] S. A. Licorish and S. G. MacDonell, "Personality profiles of global software developers," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, 2014.

[21] N. Katira, L. Williams, E. Wiebe, C. Miller, S. Balik and E. Gehringer, "On understanding compatibility of student pair programmers," in *ACM SIGCSE Bulletin*, 2004.

[22] B.-C. Catherine and D. D. Wheeler, "The Myers-Briggs personality type and its relationship to computer programming," *Journal of Research on Computing in Education,* vol. 26, pp. 358-370, 1994.

[23] E. M. Rogers, Diffusion of innovations, Simon and Schuster, 2010.

[24] S. Xiao, J. Witschey and E. Murphy-Hill, "Social influences on secure development tool adoption: why security tools spread," in *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing*, 2014.

[25] L. A. Meyerovich and A. S. Rabkin, "Socio-PLT: Principles for programming language adoption," in *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, 2012.

[26] L. A. Meyerovich and A. S. Rabkin, "Empirical Analysis of Programming Language Adoption," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2013.

[27] B. W. Boehm, Software engineering economics, Prentice-Hall, 1981.