Groktools – Tools to understand code better

a submission for Georgia Tech's Ed Tech course (CS6460)

Vinod Dinakaran
Department of Computer Science
Georgia Institute of Technology
Atlanta, GA, US
vdinakaran3@gatech.edu

***Abstract*—Groktools aims to improve the state of the art in program comprehension with tools that can be easily adopted by developers. A literature study was conducted to review the current state-of-the-art and specific gaps were identified, namely, that practitioners didn't use the result of research and that there was need for tools to: record the thought process at the time of authoring code for use later, maintain a state of focus during authoring or comprehension, record usage of the application during comprehension, identify original authors so knowledge sharing can happen and ways to avoid deep comprehension. Based on these findings, tools were proposed, built and released to the public, with some feedback gathered. In addition, a website (www.groktools.org) was created to record the project's progress as well as to foster future collection of tools that aid program comprehension. Next steps include collecting feedback from users at large, enhancing the tools to make them cross-platform and/or cross-IDE and enhancing the website to make it a true database of program comprehension tools.**

Keywords—program comprehension, tools, flow, git, atom, editor, IDE.

I. Introduction

How do developers learn and understand code? How does the newest member on a development team learn the architecture, design and idiosyncrasies of the systems that the team owns? And how can a tool – specifically the editor – be used to aid this process?

These were the original questions that kickstarted this project. As the initial exploration progressed, a final dynamic was added to this set of questions: How can we help developers craft code that is easier to comprehend to begin with?

This completed the spectrum of survey from the creation to subsequent comprehension. The focus on tools meant that the state-of-the-art was to be explored, gaps identified, solutions proposed and tools built. The rest of this paper presents each of these steps in that exact order, starting with literature survey.

At the end of this project, however, I'm left with a sense of a beginning instead of a conclusion; the tools are built, but could be better, more fully featured and built on more IDEs and editors. More importantly, however, the task of cataloguing available tools in a manner more accessible to developers seems a worthy extension, which resulted in the creation of a website that begins that task. These future motivations, the website and its future directions are covered in the final section - Next steps.

II. Literature survey

In line with the set of questions to be answered, the literature survey was undertaken in two pieces: one focused on tools and techniques that help write code such that it is easy to understand and another focused on understanding programs after they're written.

A. *How can we help developers craft code that is easier to comprehend to begin with?*

Academia and industrial practice have put forth a number of tools and techniques to help developers craft code that is easier to understand. They can broadly be described as a spectrum based on impact and cost of change (Fig 1).

On one end is the actual act of coding and on the other are industry-wide changes, with changes to the design, architecture, development lifecycle, team and organization in

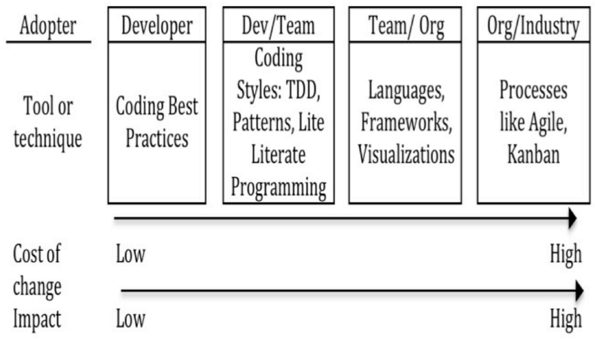
Adopter
Developer
Dev/Team
Team/ Org
Org/Industry
Tool or technique
Coding Best Practices
Coding Styles: TDD, Patterns, Lite Literate Programming
Languages, Frameworks, Visualizations
Processes like Agile, Kanban
Cost of change
Low
High
Impact
Low
High

Fig 1

between.

Closest to the act of writing code are **coding best practices** and developer tools. Coding standards and conventions for using the right formatting, appropriate names and pertinent comments are the simplest yet most effective mechanism to create readable code per Maalej, Tiarks, Roehm and Koschke[1]. Samples of such standards abound from academia [2], industry [3] and practitioners [4]. These standards are automated by code linters such as lint [5] and style checkers, while IDE features such as refactoring help to speed up the more manual tasks. Dashboards such as SonarQube[8] and Xradar[9] allow tracking progress on key metrics over time.

The next rung of the spectrum - **coding styles** like Test Driven Development [10] and Lightweight * Literate Programming [11] - are practiced by individuals but need more effort and some social acceptance to be effectively adopted. They help comprehension by reifying more of the programmers mental model into the codebase and are usually accompanied by tools to automate the reification or subsequent steps. Examples are testing frameworks that are available for most languages[13], side-by-side documentation tools like Docco [14] and full tools such as literate-programming[15]. Unrelated, but at the same level are tools that augment programming in some way: examples are debuggers and live coding environments such as Lighttable [16] that help the programmer understand the code by running it as it is being built.

At the next level of the spectrum are **languages and frameworks**, which needs a higher degree of effort and team/organizational acceptance for adoption.

- **Languages**: The past decade has seen the mainstream acceptance of functional programming concepts into mainstream languages [18], entirely new functional languages such as Clojure[19] and Elm [20], and Domain Specific Languages [21]. These languages provide a higher degree of freedom of expression that was not present before and enable the creation of less verbose, more domain-focused and understandable code.
- **Frameworks/Models**: Traditional modularization models such as Encapsulation and Object Orientation have been complemented by scalable architecture models such as Map-Reduce [22], Hardware virtualization[23] and the recent thrust towards Microservices[24] riding on the back of Container Technology . Similarly, there have been advances in UI technology to handle the multitude of devices and required speed of updates to make UIs responsive[27] and reactive[28]. While none of these are explicitly about code comprehension, they partition the mental landscape horizontally so that developers can focus on understanding a standard solution to a problem and build their applications on top of it.

Academia has put together a wide array of tools such as Software Cartography[29], Code City[30] and SHrimP[31] to aid with the comprehension of such languages and frameworks. A partial survey of such tools is available at Storey, 200532. Similarly, software vendors have created visualization and management tools to enable easy understanding of their frameworks, two non-standard examples of which would be Cloudera's Hadoop IDE[33] and Visual Devops's Cloud provisioning and management tools [34]. These particular ones were picked above regular tools such as SourceInsight[35] to show the prevalence of such tools far beyond the regular areas where comprehension is expected to happen.

* Not to be confused with older tools, which follow Donald Knuth's original idea [12]

Finally, there are process changes that need industry-wide acceptance for adoption. The move to Agile/Lean/Kanban ways from the traditional Waterfall methodology has had the effect of forcing a "ship your organization" model [36]. Aside from the benefits of incremental development making code comprehension easier, the model also makes understanding code analogous to understanding the organization structure, further concretizing the programmer's mental mode of the code.

One project worth mentioning at this stage is Alan Kay's Fundamental new computing technologies project [37], which is attempting to build an operating system in fewer than 20,000 lines of code. Regardless of the outcome of the project, the goal, the people involved and the insights gained are likely to seep into practice in the future, much as Kay's original Smalltalk language influenced the growth of Object Orientation in the 80s and 90s.

B. How do developers actually understand code?

Theoretical models proposed for code comprehension are catalogued by Storey [31]. More recently, two separate studies have tackled this question from differing angles.

The first of the studies by Maalej, Tiarks, Roehm and Koschke [1] combined empirical observation and surveys of how developers undertook program comprehension tasks, which resulted in 28 findings, summarized below:

1. Developers usually follow a recurring, structured comprehension strategy depending on the context, which includes the type of task, developer personality, the amount of previous knowledge about the application, and the type of application.
2. For tasks including changing source code, developers employ a work pattern with three steps: (1) Identification of the problem (in case of bug fixing) or identification of code locations as starting points (in case of feature implementation), (2) searching for and applying a solution, and (3) testing the correctness of the solution.
3. Developers interact with the application user interface to test whether the application behaves as expected and to find starting points for further inspection.
4. To comprehend programs, developers need to acquire runtime information. For this developers frequently execute the application using a debugger.
5. Developers try to avoid comprehension by cloning pieces of code if they cannot comprehend all possible consequences of changes.
6. Developers prefer a unified documentation structure to simplify finding information needed for comprehension.
7. Developers usually want to get their tasks done rather than comprehend software.
8. Experience of developers plays an important role in program comprehension activities and helps to identify starting points for further inspection and to filter out code locations that are irrelevant for the current task.

9. Developers comprehend software by asking and answering questions and establishing and testing hypotheses about application behavior.
10. Some developers use transient notes as comprehension support. This externalized knowledge is only used personally. It is neither archived nor reused.
11. Industry developers do not use dedicated program comprehension tools developed by the research community.
12. During comprehension tasks, IDE and specialized tools are used in parallel by developers, despite the fact that the IDE provides similar features.
13. Some developers use the compiler to elicit structural information such as dependencies and usage locations of code elements.
14. Developers do not know some standard features of tools.
15. Source code is a more trusted source of information than written documentation, mainly because documentation is often non-existent or outdated.
16. Communication with colleagues is a more important channel to access knowledge than written documentation because written documentation is non-existent and some developers prefer direct communication to writing documentation. The advantage of direct communication is that answers can be tailored to the knowledge seekers whereas the advantage of written documentation is that it is reusable.
17. Standardization – the consistent use of naming conventions and a common architecture – allows developers to become familiar with an application quickly and makes program comprehension activities easier and faster.
18. Cryptic, meaningless names hamper understanding of a piece of code.
19. Naming conventions can help to mitigate this effect but if they are too complicated they can have a negative effect.
20. Knowledge about implementation rationale and intended ways of using a piece of code helps to comprehend code but this information is rarely documented.
21. There is a gap between the interest of developers in this information and the lack of documenting it for their own code.
22. The way in which end users use an application is helpful context information in program comprehension.
23. In many cases this information is missing.
24. Developers heavily rely on communication and personal experiences for program comprehension tasks. However, only developers working for large organizations encounter problems in knowing who has this experience.
25. Developers' open-source experience and the size of their employers influence their knowledge sharing behavior.
26. The informal channels such as personal communication and comments are more popular to share knowledge about programs than the formal channels such as knowledge management tools and mailing lists.
27. While some channels are preferred to access knowledge by developers, others are rather preferred to share knowledge.
28. The more people developers know, the easier they can identify which experience is worth sharing, with whom, and how.

The second set of studies is due to Bruce Parnin and Spencer Rugaber [68-69], who took a more internal view of comprehension, i.e, as the set of events within the developer's head as comprehension (and the related interruption thereof) happens. The original research is expressed primarily in the context of code creation, not its comprehension post-fact, but is similar enough to the former scenario because code creation also requires comprehension of existing code so that changes can be made. The findings from these can be summarized thus:

1. Programmers take 10-15 minutes to resume coding once interrupted
2. Programmers are likely to get one uninterrupted 2-hour session every day
3. The following tasks were most sensitive to interruptions: during multiple concurrent edits, during navigation or search, while comprehending control and data flow of the code, and when the IDE is out of focus.

III. Problems chosen for tool development

The findings listed above were analyzed and those that are relevant to tool enhancements are:

1. Gap between research and practice: Developers don't use (or even know about) advances in code comprehension research.
2. Context Awareness in Code comprehension: Understanding code needs a record of the state at the time of building the code, not at the time of reading it. Developers are Ok to share this detail anonymously.
3. User Behavior and Usage Management: Developers sometimes pretend to be users to understand a system and sometimes avoid deep code comprehension.
4. Knowledge sharing dilemmas: Developers need to know who built the existing code.
5. Interruption Support: Developers need support to help cope with memory failures caused by interupptions.

These therefore, were the specific areas that my project aimed to address.

IV. Solutions proposed

A. *A model for code comprehension and its tooling*

To put code comprehension and it's tooling in perspective, its useful to have a mental model of the steps (Fig 2). Code comprehension occurs at two key junctures in the software lifecycle. This process is dubbed "Grok" in the diagram below, following the developer parlance for "understand" (see the Hacker's Dictionary [71]).

- A developer understands(groks) the requirements, synthesizes an architecture/design and creates the software (codes it). The only of part this intense process that's available for future inspection is the code itself.
    - There is a lot of tooling for the code process.
    - There is very little tooling for the grok process or to retain its artifacts (i.e., the understanding gained). Instead, engineering best practices and documentation (off-codebase and inline) are relied upon.
- Subsequently, a change needs to be made to the existing code – to fix a bug or to add a feature. At this point, another (or sometimes the same) developer reads the code and understands it (re-groks it, so to speak) enough to make the appropriate set of changes (codes furthers).
    - A lot of the code comprehension and visualization tooling is aimed at this juncture, attempting to reverse the initial process to the extent possible.
- The gap in understanding between the two developers where there is opportunity for the first to help (i.e., guide) the second is usually relegated to software best practices such as good coding styles and documentation. This is depicted as a gap - the black dashed arrow - above.

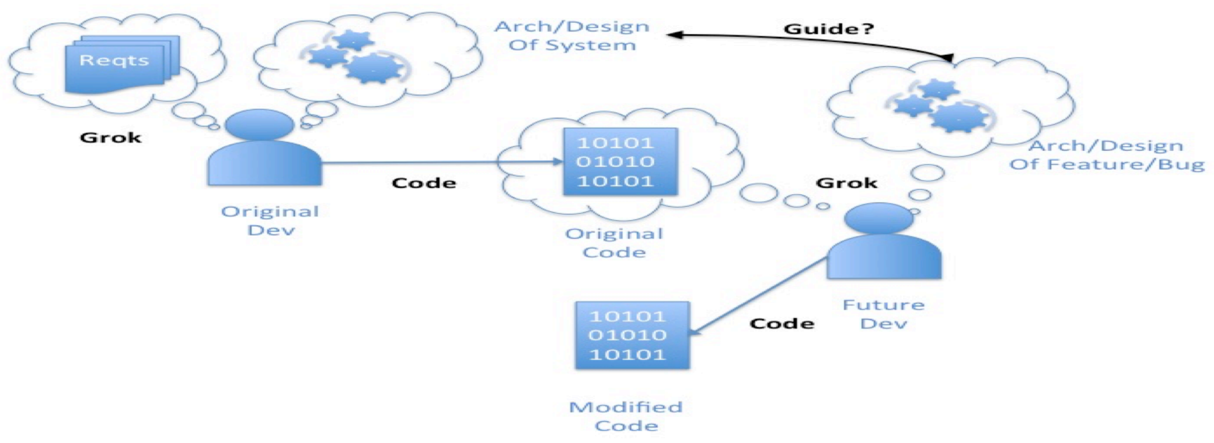

Fig 2

B. *Natural approaches to solving these problems*

Given the problem areas mentioned above, the natural next step is to propose some general approaches that might work, like so:

Problem	**Proposed Approach**
Gap between research and practice	Use this fact as a guiding principle in coming up with tools. If it's entirely a new solution, it may not work. Build a set of tools that meet the intent, but are small and can be adopted easily
Context Awareness and Code comprehension	Record developer activity and make it available for self-analysis, summarization and sharing – with respect to the developer's privacy and comfort level.
User Behavior and Usage Management	Enable automation of this step, potentially recording trial runs for replay later. Provide ability to zoom into the appropriate code easily.
Knowledge sharing dilemmas	Make the authors of code easily accessible on a per-component basis.
Interruption Support	Provide a recall/replay option so that when the developer returns to work she can get back to the zone using recall.

Clearly, none of these suggestions are particularly new and have found expression in multiple tools in some shape or form, which will be explored in the next major section.

C. *Existing Solutions*

- **Tools that make adoption easy**: Editors are generic tools by nature and therefore usually make adoption easy by sticking to standard UI affordances and open data formats.

Advanced ones are similar to IDEs in that they allow new features to be plugged into the code via extensions or plugins. The nuance in this case, therefore, is one of the ease of adoption of plugins themselves. If the plugin marketplace is crowded, developers experience the same adoption issues at the plugin level as they'd experienced at the tool level. Some IDEs such as Intellij circumvent this issue by making some features part of the core product itself. Another option for captive audiences such as .net is to keep innovating on the single IDE in the environment, namely Visual Studio.

- **Tools that record developer activity**: Minnelli, Mochi, Lanza and Kobayashi [70] describe two such tools: the DFLOW plugin for the Smalltalk Pharo IDE and the PLOG plugin for Eclipse.
- **Tools that track trial runs of the application**: Light Table by Chris Granger [16] and the Elm Language by Evan Czaplicki [20] are two examples of tools that provide rich feedback of the running application. However, they are language-specific and do not allow recording of runs for future use.
- **Tools that provide information about authors**: This is a easily satisfied ask and most VCS tools and editors have some kind of support for display of author information. For example, git-blame and Atom's git blame-based gutter display provide this information. Gource by Cauldwell [72] is another tool that provides a video narrative of authors arriving, influencing and departing from a codebase.

The key issues that make these tools less-than-optimal, therefore, are:

- **Exclusive, not inclusive solutions**: Some tools work only within their domain and do not allow for the multi-tool usage described by Maalej et al.
- **Alternative, not mainstream**: Some tools like the DFLOW plugin are built for the Smalltalk IDE, not for mainstream languages like Java, Javascript and C#.
- **Non-portable**: Most solutions are not intended to work the same across multiple Operating Systems.
- **Large, not small**: Most tools are part of behemoth sovereign applications and therefore make adoption difficult, e.g., the Eclipse Journal plugin is lost in the sea of other eclipse plugins.
- **Do not guide**: Most importantly, most existing tools do not address the gap between the initial developer and the subsequent one. The ability to guide via the codebase is therefore lost.

D. *Tools proposed, therefore*

My proposal to improve the state of response to these problems is a set of tools that I call "GrokTools", guided by the approach to address the adoption problem. For each of the other problems, at least one tool is suggested, sometimes two. Each tool is further broken up into versions that are Fallback, Comfortable or Preferred; which represent positions of implementation completion that would progressively improve the degree to which the problem is addressed.

Table 2: Proposed Solutions

Problem	**Proposed Solutions**		**Type of Solution**
	Tool #	**Description**	
Gap between research and practice	All	Create individual plugins	Fallback
		Add tools as a package of plugins for easy adoption	Comfortable
		Add tool as core feature of IDE	Preferred
Context Awareness and code comprehension	1	Same as below but less portable or IDE-specific tracker	Fallback
		Create system-wide, IDE-agnostic, portable tracker	Preferred
	2	Create a journal to capture thoughts and pre-commit decisions	Comfortable
User Behavior	3	Create a basic input record/replay tool	Comfortable
		Create a full-featured input record/replay tool	Preferred
Usage management	4	Store a reason for the change at time of commit	Preferred
Knowledge sharing dilemmas	5	Display author list for each file	Fallback
Interruption Support	6	Build a flow bar that helps resume after interruption	Preferred

To address the issues with existing tools, my project intended to:

- Implement the tools as OS-native applications where applicable and breaking them into smaller client-server architectures where applicable. This increases inclusion and portability and makes the sizes smaller.
- Implement the tools as plugins for Atom.io, Intellij IDEA and Eclipse – in that order. This addresses the mainstream developer tools.
- Most importantly, it introduces the ability for the original developer to pass information onto the subsequent ones through two mechanisms – the journal

and the commit message – improving the guidance given. To aid adoption, I plan to do this in a code-adjacent fashion.

V. Tools Built

To meet these goals, the following tools were built:

1. Lilbro: A userspace tracker for everything you do on your computer while you code.
2. Journal: A simple tool to track your thoughts as you code.
3. Recorder: The script tool that allows you to record your interactions with the code you're trying to understand so that you can replay it later for recall or to share with others.
4. whyCommit: A scripting package that adds a template for your commit messages that helps you to guide future readers so that they understand your change better.
5. Git-authors: An Atom plugin that displays the authors for a file so that you can contact them if required. Their % contributions are also shown.
6. Flozone: An Atom plugin that helps you achieve flow state (aka "the Zone") and stay there through interruptions.

In addition, a website – www.groktools.org - was created to showcase the tools as well as to continue the collection and dissemination of knowledge related to program comprehension.

VI. Tool Design and Implementation

While full source and documentation for each tool are available in the corresponding Github repository, a summary is provided below to show how the goals and intents of the project listed in IV.D are met.

lilbro has three logical components:

- A server that exposes an HTTP endpoint `/track` that allows you to send it data, which it dutifully stores in a log file. This repo contains the server code.
- Numerous clients that send the server data, each housed in its own repo under the `groktools` organization. Clients could be:
    - IDE plugins
    - Browser plugins
    - OS-based keyboard and mouse hooks
    - Anything that you'd want to capture.
    - Aggregators that make sense out of all this collected data.

To keep things above board, it is not a stealth tracker but an overt, user-space one. That is, you have to explicitly start and stop it. lilbro is built in Go and released as Linux, OSX and Windows executables for easy adoption and use.

Journal allows you to write memos to yourself (and potentially other team members) to explain the concepts that go into your code better. It is implemented as an Atom Plugin.

Recorder aims to make it easy to record and replay developer interactions with the system under study to make comprehension easier. The dream record/replay tool would allow you to record and replay, something like so:

```
./record --context="Booking ticket" --all-input -o booking_ticket.log
```

...which could be played back like so:

```
./replay booking_ticket.log
```

However, things are not that easy in the real world:

- Privacy/Security concerns: Features such as Keyboard and mouse logging are typically fraught with privacy and security concerns. OSs therefore lock these down, rightfully.
- OS constraints: Since these are typically low level features, each OS implements access to them differently, making integration and abstraction difficult.
- Brittle Solution: Assuming we get past these hurdles, we're still left with a very brittle solution: recorded data are vey specific to the conditions at the time recording and perfect replay requires a perfect recreation of those conditions.
- Aid comprehension: Assuming we're able to make a flexible solution, it must still enable comprehension for it to be of use to the developer.

Since there are no tools already, especially cross-platform, the suggestion is to use OS-native tools.

whyCommit contains tools to narrow the gap between the original author and current owner. Included are:

- A commit message template (`whycommit-template.txt`) that requires the author of the change to answer this question for future readers to grok.
- A bash/bat shell script to install the message template for one repo or for all of them. A bash/bat shell script to install a pre-commit hook preventing commits without a Why section (with an escape hatch for "small" commits detailed below).

Git-Authors is an Atom plugin that displays the authors for a file so that you can contact them if required. Their % contributions are also shown.

Finally, **Flozone** is an Atom toolbar that uses the familiar media player metaphor to help you remember what you're doing and help you tide through interruptions. flozone is a toggle-able toolbar that sits on top of your Atom window and allows you to start on a particular task by naming it. You continue to work on your task until you're done, registering it by hitting the stop button or typing a new task.

If/When you're interrupted in the middle of your task, you 'pause' the task by hitting the Pause button on the flow bar, and once the pesky interruption is done, you're reminded of your task because its name is front and center, and you can get back to the zone by pressing the Rewind button. This rewinds the last of your 15 actions in the IDE with enough delay for you to follow along, and then fast-forwards to your latest change by redoing all of the changes it just undid.

VII. Challenges/Lessons Learnt

During the execution of the project, these were the challenges faced:

1. Contributing to open source projects has unforseen complications. For example:
   a. I planned to contrinbute the whyCommit template to the git-gui project. However, its not really maintained anymore
   b. I planned to contribute a plugin to Ecplise for whyCommit but learnt there're deep dependencies between that codebase and others that honoring a git statndard feature was not met.
2. Building low-level tools such as the recorder for all operating systems is not just hard, but also fraught with security and privacy concerns.

VIII. Next Steps

1. Harden the tools: While the tools are built and published/released, they still have some testing, hardening and rounding up with features that are required.
2. Market the tools at dev venues: Without any marketing I had one user for Flozone, who downloaded it, used it and gave feedback. The plan going forward is to market it in various fora that are developer focused.
3. Promote the website: similarly, I hope the website – www.groktools.org - that currently has a project-centric focus is converted into a database of program comprehension links or programs themselves.

Acknowledgment

My mentor – Micheal Lustig – has been very helpful with feedback on the incremental reports, but his greatest help and contribution to the project was to pose the question of how the state-of-the-art could be improved by capturing as much knowledge about the code up-front instead of trying to reconstruct that condition post-fact. This was indeed a useful and fruitful diversion to have taken.

References

[1] Walid Maalej, Rebecca Tiarks, Tobias Roehm, Rainer Koschke. (Aug 2014). On the comprehension of Program Comprehension. ACM Transactions on Software Engineering and Methodology, Vol. 23, No. 4, Article 31.Retrieved from: https://mobis.informatik.uni-hamburg.de/wp-content/uploads/2014/06/TOSEM-Maalej-Comprehension-PrePrint2.pdf

[2] Mike Jackson (Dec 2012). Writing readable source code. Retrieved from: http://software.ac.uk/resources/guides/writing-readable-source-code

[3] Sandra Minjoe (n.d) Writing code that lasts. Retrieved from: http://www2.sas.com/proceedings/sugi29/134-29.pdf

[4] Wiki wiki web (July 2013). Retrieved from: http://c2.com/cgi-bin/wiki?CodingStandardList

[5] Jochen Pohl (May 2001). Lint manual. Retrieved from: http://www.unix.com/man-page/FreeBSD/1/lint

[6] PMD (Dec 2015). PMD – source code analyzer. Retrieved from: https://pmd.github.io/

[7] Checkstyle (Jan 2016) Retrieved from: http://checkstyle.sourceforge.net

[8] SonarQube (Jan 2016). Retrieved from: http://www.sonarqube.org/

[9] Xradar (2009). Retrieved from: http://xradar.sourceforge.net/

[10] Kent Beck (2002).Test Driven Development: By Example, Addison-Wesley Longman, 2002, ISBN 0-321-14653-0, ISBN 978-0321146533

[11] Github(2016). Search for term "literate programming". Retrieved from: https://github.com/search?utf8=%E2%9C%93&q=literate+programming

[12] Donald E Knuth (1992). Literate Programming. Retrieved from: http://www-cs-faculty.stanford.edu/~uno/lp.html

[13] Wikipedia. (n.d). List of known automation testing. https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

[14] Jeremy Ashkenas (2015). Docco. Retrieved from: http://jashkenas.github.io/docco/

[15] James Taylor (2014). Literate programming in JavaScript. Retrieved from: https://github.com/jostylr/literate-programming

[16] Chris Granger(Apr 2012). Light Table – a new IDE concept. Retrieved from: http://www.chris-granger.com/2012/04/12/light-table---a-new-ide-concept/

[17] Oracle (2015). Lambda Expressions. Retrieved from: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

[18] Microsoft (n.d). Functional programming in F# and C#. Retrieved from: https://msdn.microsoft.com/library/hh297108%28v=vs.100%29.aspx

[19] Rich Hickey (2008). Clojure. Retrieved from: http://www.infoq.com/presentations/hickey-clojure

[20] Evan Czaplicki (2012). Elm Language. Retrieved from: http://elm-lang.org/

[21] Martin Fowler (2008). Domain specific languages. Retrieved from: http://martinfowler.com/bliki/DomainSpecificLanguage.html

[22] Jeffery Dean and Sanjay Ghemawat (2003). Map Reduce: Simplified Data Processing on Large Clusters. Retrieved from: http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

[23] George Dunlap (2012). An introduction to full virtualization with Xen (Part1). Retrieved from: https://www.linux.com/news/enterprise/systems-management/655446-an-introduction-to-paravirtualization-and-xen

[24] Martin Fowler (Mar 2014). Microservices. Retrieved from: http://martinfowler.com/articles/microservices.html

[25] Beiderman et al (Aug 2008). Linux Containers. Retrieved from: http://linuxcontainers.org/

[26] Docker Inc. (2013). Docker. Retrieved from: https://www.docker.com/

[27] Ethan Marcotte (2010). Responsive Web Design. Retrieved from: http://alistapart.com/article/responsive-web-design

[28] Andre Staltz (2014). The introduction to Reactive programming you've been missing. Retrieved from: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

[29] David Erni (2010). Improving the mental model of software developers through Cartographic Visualization. Retrieved from: http://scg.unibe.ch/archive/masters/Erni10a.pdf

[30] Richard Wettel (2008). Visual exploration of Large-scale system Evolution. Retrieved from: http://wettel.github.io/download/Wettel08d-wcre.pdf

[31] Margaret-Anne Storey (2001). SHriMP views: an interactive environment for visualizing Java programs. Retrieved from: http://cs.brown.edu/research/softvis/papers/storey.pdf

[32] Storey, M.(2005). Theories, methods and tools in program comprehension: past, present and future. 13th International Workshop on Program Comprehension, 2005. IWPC 2005 Proceedings. Pages: 181 - 191, DOI: 10.1109/WPC.2005.38

[33] Cloudera. (2010)Hadoop User Experience. Retrieved from: http://www.gethue.com

[34] MadeiraCloud (2014). Visual Devops. Retrieved from: http://www.visualops.io/

[35] Source Dynamics, (n.d). Source Insight. Retrieved from: http://www.sourceinsight.com/

[36] Mel Conway (1968). How do committees invent? Retrieved from: http://www.melconway.com/Home/Committees_Paper.html

[37] Dan Amelang, Bert Freudenberg, Ted Kaehler, Alan Kay, Stephen Murrell, Yoshiki Ohshima, Ian Piumarta, Kim Rose, Scott Wallace, Alessandro Warth, Takashi Yamamiya (Oct 2011). Steps towards expressive programming systems. Retrieved from: http://www.vpri.org/pdf/tr2011004_steps11.pdf

[38] Roberto Minelli et el. I know what you did last summer. An investigation of how developers spend their time. (Online): http://www.inf.usi.ch/faculty/lanza/Downloads/Mine2015b.pdf

[39] Stuart E. Dreyfus. (2004) The Five-Stage Model of Adult Skill Acquisition. Bulletin of Science Technology & Society 2004 24: 177. DOI: 10.1177/0270467604264992

[40] Erik Dietrich (Sep 2012). How developers stop learning: The rise of the Expert beginner. Retrieved from: http://www.daedtech.com/how-developers-stop-learning-rise-of-the-expert-beginner

[41] Neeraj Sangal, Ev Jordan, Vineet Sinha, Daniel Jackson (2005). Using dependency models to manage complex software architecture. Retrieved from: http://sdg.csail.mit.edu/pubs/2005/oopsla05-dsm.pdf

[42] IBM (Mar 2004). Structural Analysis for Java. Retrieved from: http://web.archive.org/web/20040313014412/http://www.alphaworks.ibm.com/tech/sa4j . Note: This page and tool are now not available anymore. Some screenshots are available here: http://www.slideshare.net/scompagnone/sa4j-xerces and here: http://www.cse.unr.edu/~mbilgi/dox/sewa05.pdf

[43] Headway (2001). Headway reView. This tool is no longer available. See a screenshot at page 2 of: https://www.tomsawyer.com/casestudies/headwaysoftware.pdf

[44] Headway (2006). Structure 101. Retrieved from: http://structure101.com/

[45] Leslie Lamport, John Matthews, Mark Tuttle, and Yuan Yu (2002).Specifying and Verifying Systems with TLA+ .Proceedings of the Tenth ACM SIGOPS European Workshop (2002), 45-48.

[46] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff (2015). How Amazon Web Services Uses Formal Methods. Communications of the ACM, Vol. 58 No. 4, Pages 66-73.

[47] Wolfram Research (n.d). Mathematica. Retrieved from: https://www.wolfram.com/mathematica/

[48] R project (n.d). The R Project. Retrieved from: https://www.r-project.org/

[49] Jetbrains (2014). Resharper. Retrieved from: https://www.jetbrains.com/resharper/

[50] Andreas Gomolka, Bernhard Humm (2013). Structure Editors – old hat or new future? Retrieved from: https://www.fbi.h-da.de/fileadmin/personal/b.humm/Publikationen/Gomolka_Humm_-_Structure_Editors__Springer_ENASE_.pdf

[51] Bran Selic (2003). IEEE Software archive. Volume 20 Issue 5, September 2003. Page 19-25

[52] John Gruber (2004). Markdown. Retrieved from: http://daringfireball.net/projects/markdown/

[53] Hong Xu et al (n.d). Editor Config. Retrieved from: http://editorconfig.org/

[54] Gina Trapani (2009). Todo.txt CLI manages your tasks from the command line. Retrieved from: http://lifehacker.com/5155450/todotxt-cli-manages-your-tasks-from-the-command-line

[55] John Weigley (2003). Ledger CLI. Retrieved from: http://www.ledger-cli.org/

[56] Roedy Green (2004). Java Source code SCID-style Editor. Retrieved from: http://web.archive.org/web/20040702074323/http://mindprod.com/projects/scid.html

[57] Scott Stanchfield (2007). Why Visual Age for Java. Sec titled “A repository”. Retrieved from: http://javadude.com/articles/whyvaj.html

[58] Oracle (2014). Processing Data with Java 8 Streams, Part 1. Retrieved from: http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

[59] Brendan Considine (2014). The inspector connection, Issue #1, Migration Translation. Feature #6. Retrieved from: http://blog.jetbrains.com/idea/2014/09/the-inspection-connection-issue-1/

[60] Andrew Gerradine (Jan 2013). Go fmt your code. Retrieved from: https://blog.golang.org/go-fmt-your-code

[61] Mozilla (n.d). Collections. Retrieved from: https://addons.mozilla.org/en-US/firefox/collections/

[62] Neil Pappalardo et al (1966). Mumps. No paper available. A tutorial was retrieved from: http://www.cs.uni.edu/~okane/source/MUMPS-MDH/MumpsTutorial.pdf

[63] IBM (1979-date). Transaction Processing Facility. Retrieved from: http://www-03.ibm.com/software/products/en/ztransaction-processing-facility

[64] Eric S Raymond (Aug 2000). Homesteading the Noosphere. Sec 5. Retrieved from: http://www.catb.org/esr/writings/cathedral-bazaar/homesteading/

[65] Robert C Martin et al (2009). Manifesto for Software Craftsmanship. Retrieved from: http://manifesto.softwarecraftsmanship.org/

[66] AOSABook (n.d). The architecture of Open Source Applications. Vols I and II. Retrieved from: http://aosabook.org/en/index.html

[67] Raymond Chen (n.d). The old new Thing. Retrieved from: https://blogs.msdn.microsoft.com/oldnewthing/

[68] Bruce Parnin, Spencer Rugabrer (2010). Resumption strategies for interrupted programmer tasks. Retrieved from http://ieeexplore.ieee.org.prx.library.gatech.edu/stamp/stamp.jsp?tp=&arnumber=5090030

[69] Bruce Parnin, Spencer Rugaber (2012) Programmer Information needs after mmory failure. Retrived from http://ieeexplore.ieee.org.prx.library.gatech.edu/stamp/stamp.jsp?tp=&arnumber=6240479

[70] Roberto Minnelli, Andrea Mocci, Michele Lanza and Takashi Kobayashi (May 2015). I know what you did last summer. An investigation of how developers spend their time. Retrieved from: *http://www.inf.usi.ch/faculty/lanza/Downloads/Mine2015b.pdf*

[71] Robert A Heinlein. “Grok”. Retrieved from: http://www.hackersdictionary.com/html/entry/grok.html

[72] Andrew Cauldwell (2014). Gource. Retrieved from http://gource.io.