

MASTER

Improving the usability of the domain-specific language editors using artificial intelligence

Rajasekar, Nandini

Award date:
2021

[Link to publication](#)

Disclaimer

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Improving the usability of the domain-specific language editors using artificial intelligence

Master Thesis Preparation Report

Nandini Rajasekar

Supervisors:
Prof. Dr. M.G.J. van den Brand
M. Verano Merino, MSc

1.0

Eindhoven, July 2020

Contents

Contents	ii
List of Figures	iii
1 Introduction	1
1.1 Language Oriented Programming	1
1.2 Language Workbenches	1
1.3 Artificial Intelligence	2
2 Motivation	4
3 Goal and Approach	5
3.1 Goal	5
3.2 Research questions	5
3.3 Approach	5
4 Literature Survey	7
5 Prototype	10
5.1 Next steps	11
6 Project Planning	12
7 Conclusion	13
Bibliography	14

List of Figures

1.1	General classification of AI techniques	2
3.1	Overview of the approach	6
5.1	Logged data about a created node	10

Chapter 1

Introduction

Software engineering has a lot of advancements in the state of the art. Integrated Development Environments (IDEs) are complex applications that integrate multiple tools for creating and manipulating software project artifacts[14]. IDEs and software languages facilitate the development and maintenance of software. The user's interaction with the IDE provides a rich source of information that can help the programmers to avoid erroneous code in their development in the future. By capturing and exploiting comprehensive, fine-grained IDE interactions, we can build intelligent IDEs that improve programmer productivity[15] and also reduce the cost of software. The integration of Artificial Intelligence (AI) in IDEs can generate smart editors which can provide assistance to programmers through code completions or intelligent recommendations. In this project, we focus on improving the editor of a language workbench using the editing logs and support the developers and language engineers providing intelligent recommendations.

1.1 Language Oriented Programming

In general, software languages are classified by generality and specificity as General-purpose languages (GPL) and Domain-specific languages (DSL). A GPL is a computer language that is broadly applicable across application domains, and lacks specialized features for a particular domain in contrast to DSLs.

Language oriented programming (LOP) is a software-development paradigm where "language" is a software building block with the same status as objects, modules and components[11]. In LOP, rather than solving problems in general-purpose programming languages, the programmer creates one or more DSLs for the problem first, and solves the problem in those languages. LOP takes the approach to capture requirements in the user's terms, and then try to create an implementation language as isomorphic as possible to the user's descriptions, so that the mapping between requirements and implementation is as direct as possible. Advantages of Language Oriented Programming include separation of concerns, high development productivity, highly maintainable design, highly portable design, opportunities for reuse and user enhancable system[32]. Some tools and languages are designed to support Language oriented programming. Language workbenches(LWBs) such as JetBrains MPS, Rascal, Spoofax, Kermeta, or Xtext provide the tools to design and implement DSLs and language oriented programming. The Racket programming language is designed to support LOP[11].

1.2 Language Workbenches

Language workbenches, a term popularized by Martin Fowler in 2005[2], are tools that support the efficient definition, reuse and composition of languages and their Integrated development environments (IDEs). LWBs make the development of new languages affordable and, therefore, support a new quality of language engineering, where sets of syntactically and semantically integrated

languages can be built with comparably little effort [10]. This can lead to multi-paradigm and LOP environments that can address important software engineering challenges. LWBs exist in many different flavors, but they are united by their common goal to facilitate the development of domain-specific languages. JetBrains Meta Programming System (MPS) is a Language workbench developed by JetBrains. MPS is a tool to design domain-specific languages. It uses projectional editing which allows users to overcome the limits of language parsers, and build DSL editors, such as ones with tables and diagrams. MPS solves grammar ambiguity issues by working with the Abstract Syntax Tree (AST) directly[16]. In this project, we mainly focus on improving the editor of MPS.

1.3 Artificial Intelligence

Artificial Intelligence is defined as the “The science and engineering of making intelligent machines, especially intelligent computer programs” by John McCarthy[24]. AI is based on the principle that human intelligence can be defined in a way that machines (or computers) can easily mimic cognitive functions, learn from experience, adjust to new inputs and execute tasks that range from simple to complex. The goals of artificial intelligence include learning, reasoning, problem solving and perception. As technology advances, previous benchmarks that defined artificial intelligence become outdated.

AI technologies train computers to accomplish specific tasks by processing large amounts of data and recognizing patterns in the data. It is becoming significant as it automates repetitive learning, analyse data deeply and fully utilize data and get the most out of it. It adds intelligence to existing products and applications, making them even smarter and efficient [1].

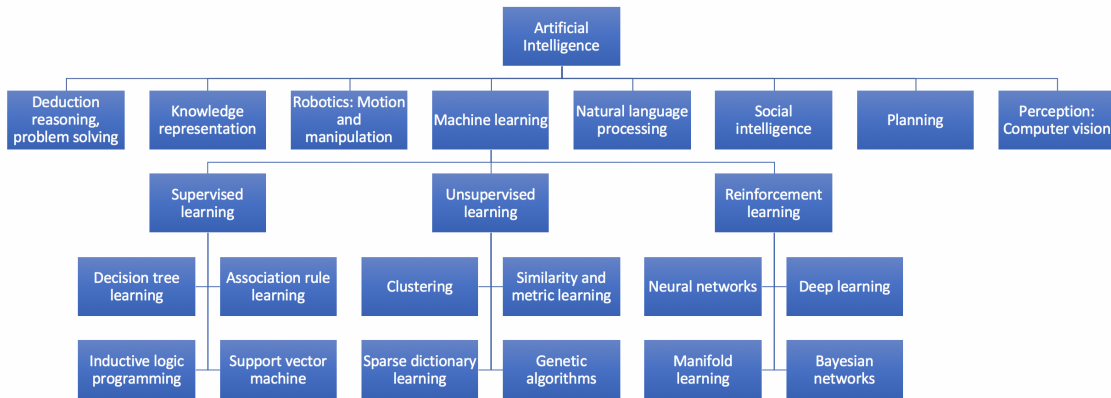


Figure 1.1: General classification of AI techniques

The applications for artificial intelligence grow day by day. Some examples of machines with artificial intelligence include computers that play chess and self-driving cars[23]. The technology can be applied to many different sectors and industries, for example it is used in the banking sector to provide customer support, detect anomalies and credit card frauds[9]. It has commercial applications for chatter-bots in online customer support and many others in the field of e-learning. It is also used in healthcare, space exploration, gaming industry, social media, marketing and many more. General classification of AI techniques is shown in the Fig.1.1[21].

There has been a recent surge in interest in the application of AI techniques to Software Engineering (SE) problems. AI algorithms and techniques also find important and effective applications that impact on almost every area of software engineering activity. In particular, the SE community

has used three broad areas of AI techniques: 1) Computational search and optimisation techniques (the field known as Search Based Software Engineering (SBSE). 2) Fuzzy and probabilistic methods for reasoning in the presence of uncertainty. 3) Classification, learning and prediction [17] AI techniques are proving to be well-suited to this changing world and also the software community for their developments. In software development, it is making the process of designing, developing, and deploying software faster, better, and cheaper. AI-powered tools aid software engineers and testers to be more productive and efficient, enabling them to produce higher-quality software faster and at lower cost.

Recommendation systems uses AI to assist software engineers or developers in development tasks such as code auto-completion and recommending likely code reviewers for a given code change. It supports by answering some typical questions about which software changes probably introduce a bug, which requirements to implement in the next software release, which method calls might be useful in the current development context and which software components (or APIs) to reuse. in the scenarios of software engineering [29].

In this project, we aim on improving the editor of MPS Language workbench using AI techniques and the suitable technique will be determined later in the project during implementation.

Chapter 2

Motivation

Domain specific languages are mostly created using the Language work benches. These LWBs support the efficient definition, reuse and composition of languages and their IDEs. LWBs make the development of new languages affordable, where sets of syntactically and semantically integrated languages can be built with comparably little effort [10]. However, these LWBs can still be improved and made smarter to improve productivity and usability.

Usability is defined as the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. The difference between usability and other software qualities is that to achieve it, one has to concentrate not on the system features but also on user-system interaction [7]. Integrated development environments (IDEs) are popular applications that serve the needs of such user system interaction for software development through editors.

Professional software developers spend approximately one third of their time working in an integrated development environment (IDE), which makes it the most used application during a working day of a software developer[12]. An advanced (context aware) intelligent editor is required to reduce the cognitive load [20], mistakes, time and effort, and to improve the development process. Such an intelligent editor can be incorporated in the IDEs using Artificial Intelligence, which in turn will benefit the LWBs.

Mining the data from the IDE editing logs, processing them using suitable AI techniques to provide assistance to the language engineers through recommendations is the main goal of this project. MPS language workbench was chosen as the IDE of interest.

Chapter 3

Goal and Approach

3.1 Goal

The goal of this project is to improve the editor in a LWB by learning from the editing logs. The logs generated from the IDE will be used as an input for a learning algorithm and the editor will be improved to produce intelligent suggestions for that language enhancing the usability of the editor. The objective is to reduce the effort, save time for language engineers, developers and to improve the quality of the editor. The research questions were formulated as shown below

3.2 Research questions

How to improve the editor in a Language workbench by learning from the editing logs?

1. What significant data can be extracted from the editing logs to realize usability improvements?
2. What are the suitable AI techniques for improving the editor in a LWB?
3. How to utilize the AI techniques for enhancing the usability of the editor?

3.3 Approach

In this project, we create a link between AI and software language engineering (SLE). To improve an editor for a DSL using AI techniques: interaction among the language, the users, the development environment, the grammar and the texts used needs to be understood. AI has several techniques that can be used for this application. The approach that is taken here is to use AI techniques like machine learning to learn the user's behaviour from their interaction with the language editor through the IDE and train the learning algorithm accordingly. The most suitable machine technique and the implementation details will be determined at a later in the project. To get feedback on the user interaction, the data obtained from the sequence of the activities performed by the user will be collected in the form of logs and given as an input to the learning algorithm. The language editor will thus be improved by providing intelligent suggestions (like Templates, auto-completion, contextual recommendations and auto navigation) helping the language engineers and the developers to program efficiently with less effort. The overview of this approach is shown in the Fig.3.1.

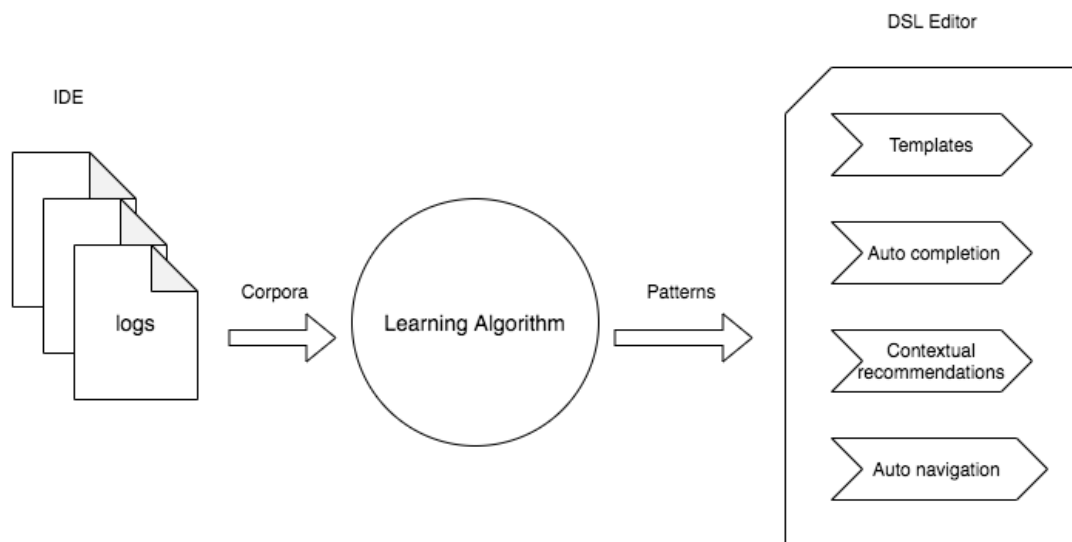


Figure 3.1: Overview of the approach

Chapter 4

Literature Survey

In this chapter, a detailed study on the related work and advancements in the IDE usability improvements is discussed.

Software engineering recommender systems assist software engineers by making recommendations such as code autocompletion and recommending likely code reviewers for a given code change. Allamanis et al. [4] made a detailed survey work on the research works carried out in the at the intersection of machine learning, programming languages, and software engineering. A taxonomy based on the underlying design principles of each model is presented. Recommender systems is one of the topics which was discussed. One of the most commonly used features of the recommender system used in IDE is Code Completion.

Most of the commonly used IDEs like Eclipse [12], Visual Studio [3] has code completion features. Amann et al [6] and Murphy et al [25] showed that code completion is one of the most used IDE features. Code completion assists developers by reducing typographic and other common errors, effectively improving developer productivity. These code completion features may include pop-ups when typing, calling methods and APIs on typed classes and objects, querying parameters of functions, variable or function name disambiguation, and program structure completion that use code context to reliably predict following code [30].

Traditional code completion tools in IDEs typically return suggestions in alphabetic order, rather than in relative order of predicted relevance to the context [4]. It also does not provide any ranking information. Hence users needs to scroll through the long alphabetically list. Many IDEs support prefix filtering to reduce the number of irrelevant recommendations. Maraoiu et al. [4] shows that developers tends to heavily rely on prefix filtering to further reduce the number of choices provided in the traditional code completion tools. This effectively help developers who knows what they are looking for, but it is of limited help for developers who are unfamiliar with the API [8]. So the main target behind the intelligent code completions is to target the needs of developers that are also unfamiliar with an API.

The main idea behind statistical code completion is to improve suggestion accuracy by learning probabilities over the suggestions. There by providing the users with a ranked list. Statistical code completion was first studied by Bruch et al. [8] who extracted features from code context to suggest completions for method invocations and constructors. Best Matching Neighbor (BMN) algorithm by Bruch et al. [8] use code repositories containing clients of an API to build a knowledge base that contains common usages for different types of that API. On a completion event, the models learned from the repositories are used to show relevant proposals.

Besides the static type of the variable on which code completion is triggered, these systems also consider some structural context of the code being developed to select the models from which to extract the recommendations. The method within which the completion system was triggered

is an example of such a structural context [27]. That stream of work evolved into the Eclipse Code Recommenders [33].

Proksch et al. [27] uses Bayesian networks as an alternative underlying model, use additional context information for more precise recommendations, and apply clustering techniques to improve model sizes. It also shows that Pattern Based Bayesian Network (PBN) can obtain comparable prediction quality to BMN, while model size and inference speed scale better with large input sizes.

The methods of Bruch et al. [8] and Proksch et al. [27] rely heavily on manually extracting features that are relevant to the completion task. For example, Proksch et al. [27] extract features such as the direct supertype of the enclosing class, the kind of the definition, etc.

Machine learning methods mentioned in [8] and [27] used feature-based models to predict the next method to be invoked. However, these models can only support suggestions on a limited, pre-defined set of APIs and do not have the capability to generalize to different APIs or different versions of the same API.

Other models, such as n-gram language models were first studied by Hindle et al. [19] who create a token-level n-gram language model completion models. This was arguably the first model that performed unconstrained code completion and was shown to work well. Also it can be called as “feature-less” machine learning models of code completion, because it does not require extraction of specific features, but instead view the code as a sequence of tokens [30]. In contrast to text, code tends to be more verbose [19], and much information is lost within the $n-1$ tokens of the context. To tackle this problem, Nguyen et al. [26] extended the standard n-gram model by annotating the code tokens with parse information that can be extracted from the currently generated sequence. As a results, this increases the available context information allowing the n-gram model to achieve better predictive performance [30].

Tu et al. [31] noticed that source code tends to have a localness property, (i.e. tokens tend to be repeated locally). The authors showed that introducing a cache-based language model, performance could be improved. Hellendoorn and Devanbu [18], then set to test neural models of code along with n-gram language models for the task. Their evaluation showed that carefully tuning an n-gram language model, with a hierarchically scoped cache, outperforms some neural models.

In [14], Gasparic et al. presented a context model that includes thirteen contextual factors, which capture various situations in which developers interact with an IDE. This context model can be used to support and enhance user interaction with an IDE or to improve the accuracy and timing of recommendations produced by recommendation systems for software engineering (RSSEs).

As mentioned before the approach used by Bruch et al. [8], Proksch et al. [27], is to extract hard-coded features from the completion context and to learn which candidate completion targets are relevant in the given context. This approach, however, misses the opportunity to learn richer features directly from data: an approach that has recently been made possible by deep learning techniques. Furthermore, such feature extraction approaches learn about each individual API and cannot generalize to unseen ones. This requires sufficient example usages of an API to learn about those features. However, in many cases, this is not possible [30].

Deep learning methods are central to “feature-less” models and eliminate the need for hand-coded features across domains (e.g. image recognition) and can directly learn from raw data [30]. Machine learning-based neural models traditionally treat code completion as a generation problem [18], [13], [22], [28] in which the model needs to generate the full completion. However, generating predictions requires learning about all potential code identifiers, which are significantly sparse, i.e.

identifiers are extremely diverse and occur infrequently [5].

In [30], it was shown that by treating the neural code completion problem as the problem of ranking the candidate completion targets returned from a static analysis yields significantly improved results and allows them to build lightweight and accurate neural code completion models. Also different trade-offs in terms of completion accuracy, memory requirements and computational cost were also studied. Because those characteristics are important for real-life deployment.

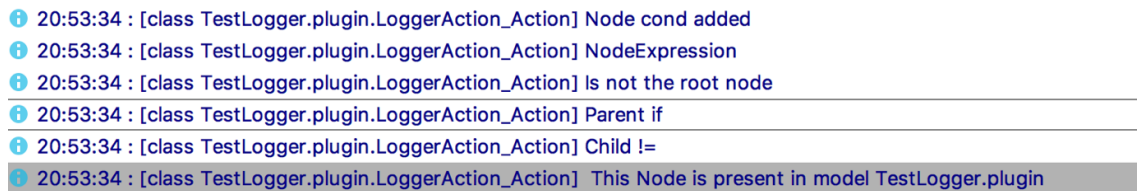
Chapter 5

Prototype

IDE logs with captured user interactions is essential to improve the editor as the user interactions helps in monitoring any abnormal or sequential activities performed by the user. However in the current MPS platform, the default logs generated by the IDE does not capture the sufficient user interaction data to monitor the activities of the user. So, a prototype logger plugin is made in MPS language workbench to capture the necessary user interactions with the editor. The captured user interactions include any changes made to the AST, the user's keystrokes in the editor and all changes made to the model in the MPS projectional editor.

A new plugin with an action in the tools menu of the MPS IDE was created. When the action is invoked the logger gets started. All the changes made to any node and the model was monitored and logged and this was achieved by the use of listeners offered by the MPS framework. The data that is logged includes a time stamp and the message of the changes made to the AST. In this way, the required data for the user interactions with the editor was gathered. Similarly, the position information of the cursor in the editor at any moment is also obtained during the user interaction. By logging these data, it was confirmed that any type of log data about the changes made in the editor that might be required for the analysis can be acquired.

Some of the logs obtained from the prototype plugin created in MPS are changes made to the nodes, models and project. Few examples of the data that can be logged for a node include its parent, child, if it is root node or not, its target concept from which it is implemented, in which model is the node present, its concept, references, property changes, creation and removal. Similar data is also obtained for models Below is an example of the data logged when a node is created in the language editor in MPS. Here, an if condition is created in the editor and the data about the if node is logged. The data delivered shows that a condition node was added and it is an expression which has a parent node called if and a child node as a condition operator. It also logs in which model the node belong. Similarly data about every node that is created, changed and removed can also be logged by the plugin.



```
i 20:53:34 : [class TestLogger.plugin.LoggerAction_Action] Node cond added
i 20:53:34 : [class TestLogger.plugin.LoggerAction_Action] NodeExpression
i 20:53:34 : [class TestLogger.plugin.LoggerAction_Action] Is not the root node
i 20:53:34 : [class TestLogger.plugin.LoggerAction_Action] Parent if
i 20:53:34 : [class TestLogger.plugin.LoggerAction_Action] Child !=
i 20:53:34 : [class TestLogger.plugin.LoggerAction_Action] This Node is present in model TestLogger.plugin
```

Figure 5.1: Logged data about a created node

5.1 Next steps

The prototype was build to study the feasibility to log and capture user's fine-grained interactions with the editor. The result obtained the prototype can be applied to build the actual project. As a following work, the implementation of the project will be an extension of the prototype discussed above. The actual use cases for which the editor improvements will be performed in MPS will be refined. The user interactions will be monitored and the necessary data will be mined to use as an input to the learning algorithm. The suitable AI technique that will be applied to enhance the usability of the editor will be chosen and implemented. Further, the result of the implementation will be validated. A detailed planning is presented in the next chapter.

Chapter 6

Project Planning

The planning of the further work of the project is presented in the table below.

No.	Content	Duration
1.	Investigate and confirm use cases and Find suitable AI technique to implement in MPS editor	0.5 months
2.	Extention of the prototype as implementation, Collection of editing logs and other necessary data Integrating AI with MPS	2.0 months
3.	Validation	0.5 months
4.	Report writing	1.0 month

Table 6.1: Planning Table

Chapter 7

Conclusion

In this report, the motivation to aid the software developers and language engineers with intelligent IDEs is discussed. Furthermore, the goal and approach to improve the editor usability in LBWs using the user's editing logs and AI techniques is presented in detail. The research questions were formulated and a literature survey was performed to study the related work in the field that addresses different techniques used for the advancements in the usability of IDEs. A prototype was built in MPS Language workbench to study and monitor the sequence of activities performed by the user in the editor. The future work and the planning for the project are also presented.

Bibliography

- [1] Artificial intelligence – what it is and why it matters. 2
- [2] Language workbenches: The killer-app for domain specific languages? 1
- [3] Visual studio ide, code editor, azure devops, app center, Jul 2020. 7
- [4] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018. 7
- [5] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216. IEEE, 2013. 9
- [6] Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. A study of visual studio usage in practice. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 124–134. IEEE, 2016. 7
- [7] Ankica Barisic, Vasco Amaral, Miguel Goulao, and Bruno Barroca. How to reach a usable dsl? moving toward a systematic evaluation. *Electronic Communications of the EASST*, 50, 2012. 4
- [8] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 213–222, 2009. 7, 8
- [9] Dahee Choi and Kyungho Lee. An artificial intelligence approach to financial fraud detection under iot environment: A survey and implementation. *Security and Communication Networks*, 2018, 2018. 2
- [10] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering*, pages 197–217. Springer, 2013. 2, 4
- [11] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, 2018. 1
- [12] Eclipse Foundation. Eclipse downloads: The eclipse foundation. 7
- [13] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. Cacheca: A cache language model based code suggestion tool. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 705–708. IEEE, 2015. 8
- [14] Marko Gasparic, Gail C Murphy, and Francesco Ricci. A context model for ide-based recommendation systems. *Journal of Systems and Software*, 128:200–219, 2017. 1, 8

- [15] Zhongxian Gu, Drew Schleck, Earl T Barr, and Zhendong Su. Capturing and exploiting ide interactions. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 83–94, 2014. 1
- [16] Slimane Hammoudi, Luís Ferreira Pires, Bran Selic, and Philippe Desfray. *Model-Driven Engineering and Software Development*. Springer, 2019. 2
- [17] Mark Harman. The role of artificial intelligence in software engineering. In *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*, pages 1–6. IEEE, 2012. 3
- [18] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773, 2017. 8
- [19] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012. 8
- [20] Reid Holmes, Tristan Ratchford, Martin P Robillard, and Robert J Walker. Automatically recommending triage decisions for pragmatic reuse tasks. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 397–408. IEEE, 2009. 4
- [21] Inknowvativeconcepts. The global race for artificial intelligence: Weighing benefits and risks, Feb 2018. 2
- [22] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code!= big vocabulary: Open-vocabulary models for source code. *arXiv preprint arXiv:2003.07914*, 2020. 8
- [23] Paul Long, Sarah Lawrey, and Victoria Ellis. *Cambridge International AS and A Level IT Coursebook with CD-ROM*. Cambridge University Press, 2016. 2
- [24] John McCarthy, Jan 1970. 2
- [25] Gail C Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the elipse ide? *IEEE software*, 23(4):76–83, 2006. 7
- [26] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 532–542, 2013. 8
- [27] Sebastian Proksch, Johannes Lerch, and Mira Mezini. Intelligent code completion with bayesian networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):1–31, 2015. 8
- [28] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428, 2014. 8
- [29] Martin Robillard, Robert Walker, and Thomas Zimmermann. Recommendation systems for software engineering. *IEEE software*, 27(4):80–86, 2009. 3
- [30] Alexey Svyatkovskoy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Franco, and Miltiadis Allamanis. Fast and memory-efficient neural code completion. *arXiv preprint arXiv:2004.13651*, 2020. 7, 8, 9
- [31] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280, 2014. 8

- [32] Martin Ward. Language oriented programming. *Software-ConceptsTools*, 15:147–161, 01 1994.
1
- [33] Webmaster. Archived projects: The eclipse foundation. 8