



## Original Research

# How to do things with code: An analytical framework for linguistic inquiry into programming environments

Elizaveta G. Grishechko<sup>1\*</sup>

<sup>1</sup>RUDN University, Moscow, Russia

Research on programming practice gives sustained analytical attention to source code and the written materials that accompany it, including comments, documentation, issue reports, and discussion threads. In these accounts, terms such as *text*, *communication*, *discourse*, *genre*, and *sublanguage* are recurrently invoked in addressing code and developer activity from a range of perspectives. Their repeated use points to a convergence with linguistic inquiry and invites consideration of code and developer activity in linguistic terms. The present study proceeds from a principle of a division of labour between fields and, to that end, identifies a baseline set of linguistic constructs appealed to in programming research to suggest directions for their investigation in line with linguistic tradition. The procedure combines systematic mapping of studies on programming practice with analytical classification grounded in distinctions established in the study of language. Following the identification of a stable set of constructs, the study clarifies the analytical capacity in which they are mobilised in programming research and advances a set of analytically grounded lines of inquiry in which these constructs may be examined in relation to textual cohesion, lexical organisation, syntactic relations, speech acts, discourse continuity, and indexical marking. The study has implications for recognising programming environments as a distinctive empirical setting for linguistic research, in which text, discourse, and communication practices are available for systematic analysis in established approaches to language study. The directions proposed here are not exhaustive and remain open to further empirical work.

**Keywords:** *source code, developer discourse, developer communication, programming communities, sublanguage, genre, programming culture*

### Article history:

Submitted October 10, 2025

Revised February 2, 2026

Accepted March 9, 2026

### CRediT Author Statement:

Elizaveta G. Grishechko: Conceptualisation, Methodology, Validation, Formal Analysis, Writing – Original Draft, Writing – Review & Editing, Visualisation.

### Conflict of interest:

The author declared no conflict of interest.

### Data availability statement:

The data supporting this study's findings are available from the corresponding author upon reasonable request.

### Funding:

This paper has been supported by RUDN University project No. 061012-0-000 'Linguistic-cognitive forecasting tools in corporate discourse'

### Doi:

10.22363/2521-442X-2026-10-1-54-71

### For citation:

Grishechko, E. G. (2026). How to do things with code: An analytical framework for linguistic inquiry into programming environments. *Training, Language and Culture*, 10(1), 54–71.



This is an open access article distributed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/) (CC BY-NC 4.0), which allows its unrestricted use for non-commercial purposes, subject to attribution. The material can be shared/adapted for non-commercial purposes if you give appropriate credit, provide a link to the license, and indicate if changes were made.

## 1. INTRODUCTION

The overwhelming majority of studies examining source code proper and natural-language artefacts produced in software development specifically has been conducted within software engineering, natural language processing, and related technical disciplines. In this literature, textual elements accompanying source code are analysed in relation to software-development tasks. Research on comment analysis develops taxonomies of documentation

comments and automated procedures for assessing comment quality and consistency with source code (Pascarella et al., 2019; Steidl et al., 2013; Yang et al., 2019), extracts programme constraints expressed in comment sentences to detect discrepancies between documentation and implementation (Tan et al., 2007), and identifies statements of technical debt in comment corpora for repository analysis (De Freitas Farias et al., 2020). Work on automated documentation recovery mines bug reports and mailing lists to

\* Corresponding author

© Elizaveta G. Grishechko 2026

Licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/)

extract descriptions of programme behaviour associated with specific methods and code fragments (Panichella et al., 2012), while neural code-summarisation systems generate short textual descriptions of source-code fragments from large collections of code—text pairs (Iyer et al., 2016). Large-scale analyses of programming-forum archives apply topic modelling and statistical classification procedures to collections of Stack Overflow posts to identify recurring programming problems and categories of technical knowledge (Barua et al., 2014; Treude et al., 2011) and develop automated summarisation systems for discussions extracted from mailing lists, chat channels, and issue trackers (Ersan, 2023). Research on source-code naming systems analyses identifier naming conventions and relations between words appearing in identifiers, comments, and code elements to improve code search, repository navigation, and automated development tools (Allamanis et al., 2015; Deißeböck & Pizka, 2006; Newman et al., 2017; Sridhara et al., 2008). Additional studies analyse comment evolution in version-control histories and examine the contribution of comments to programme comprehension during maintenance tasks (Abdelsalam et al., 2026; Fluri et al., 2007, 2009; Nielebock et al., 2018), while large-scale analyses of GitHub discussions quantify textual features of developer posts as indicators of contributor roles and collaboration behaviour within software projects (Han et al., 2024).

This extensive body of work demonstrates sustained analytical attention to textual artefacts in software systems and consistently qualifies them as integral to programming environments. Source code itself, together with the written records accompanying it (including comments, README files, issue reports, and forum discussions), undergoes systematic analytical scrutiny, with studies processing textual material as documentation data, lexical tokens, or statistical input for computational models designed to support code comprehension, documentation generation, or repository analysis. The literature therefore provides clear evidence that both source code and the natural-language materials surrounding it form a stable and indispensable element of programming environments and thus invite continued scholarly contemplation.

In this contemplation, a recurrent pattern emerges, where investigation of technical questions concerning software-development tasks repeatedly advances into a territory that may be described, with little exaggeration, as an area of linguistic pursuit. Studies of programming practice routinely mobilise terms such as *communication*, *text*, *genre* and *discourse* when addressing software-related activity or developers as communities of practice. For example, several authors refer to code itself or to developer interaction using the term *discourse* (Brock, 2016; Brock & Mehlenbacher, 2018; Ersan, 2023; Knuth, 1984; Marino, 2020). In these works, however, the term functions primarily as a

general reference to developer interaction, not as a notion grounded in linguistic discourse theory. In addition, it appears largely interchangeable with the notion of *communication* and is invoked descriptively in accounts of collaboration or in reflections on the textual qualities of code. Explicit theoretical specification of these and related linguistic constructs remains outside the scope of such studies, leaving the terminology conceptually suggestive yet analytically indeterminate.

Linguistic investigations that survey programming environments as sites of language use do exist, yet they remain relatively limited in number and temporally scattered and do not offer a systematic analytical treatment of these practices within established frameworks of linguistic inquiry. As a result, the analytical status of programming-related textual, communicative and discursive practices within linguistic research proper remains insufficiently defined.

To address this conceptual gap, the present study proposes a systematisation of linguistic constructs appearing in research on programming practice — including discussions of source code, natural-language records, developer interaction, and collective programming practice — and outlines a set of possible directions for their linguistic investigation. To that end, the study addresses the following research questions:

1. Which linguistic constructs are invoked in existing programming scholarship?
2. In what analytical capacity are these linguistic constructs mobilised?
3. Which directions of analysis may be proposed for linguistic inquiry into programming environments?

The key hypothesis of the study is that software and programming research already recognises linguistic phenomena and makes use of linguistic terminology in the description of source code and developer activity. Linguistic inquiry, in turn, may specify these constructs in disciplinary terms and propose directions for their systematic investigation. On this basis, the study sets out a point of departure for linguistic inquiry into programming environments as sites of language use, grounded in a principle of division of labour between fields. In this sense, the study is intended to contribute to the articulation of a linguistically grounded field of inquiry concerned with programming environments and the written artefacts produced in software development, placing these materials in relation to established traditions of linguistic scholarship.

## 2. METHODOLOGY

### 2.1. Analytical orientation

The study is situated at the intersection of linguistic inquiry and research on programming practice and proceeds through analytical reconstruction as its principal procedure. The material consists of research publications

addressing programming environments in which linguistic terminology is used in reference to source code, developer activity, and related written artefacts. In these studies, such designations are introduced for descriptive or technical purposes, while their specification in terms established in language research remains outside their immediate scope. This creates the conditions for a division of labour between fields, where programming research provides access to material and linguistics provides the means for its analytical specification. In this sense, analytical reconstruction refers to a procedure of theoretical specification in which terms introduced in one field are reformulated in relation to established distinctions and units in another.

## 2.2. Corpus compilation

The procedure combines elements of systematic mapping (Petersen et al., 2008), where studies are organised according to predefined analytical variables, with theory-driven classification grounded in distinctions recognised in linguistic research (Croft & Lafferty, 2003). The aim is not to code empirical content, but to register how linguistic constructs are invoked and to clarify how they may be further developed in linguistic inquiry. The corpus of reviewed studies is defined as a finite set

$$S = \{s_1, s_2, \dots, s_n\}$$

and includes  $n=84$  publications comprising research articles and monograph-length works. The corpus was compiled through targeted selection of studies addressing programming practice in which linguistic terminology is used in reference to code, developer interaction, or accompanying written artefacts. Publications were identified through database searches (Scopus, Web of Science, Google Scholar) using a fixed set of search terms, including *source code*, *code readability*, *code comments*, *software documentation*, *developer communication*, *developer discourse*, *programming discourse*, *programming culture*, *developer identity*, *GitHub discussions*, and *Stack Overflow*, followed by reference tracing. Search terms were combined iteratively, and the set was expanded until no new relevant terminological designations were identified. Search queries were implemented separately in each database using Boolean combinations of keywords, with results limited to peer-reviewed publications in English. No temporal restrictions were applied.

The corpus spans software engineering, natural language processing and machine learning, technical communication, software studies, Critical Code Studies, and linguistics. Following retrieval, publications were reviewed for relevance. The initial retrieval yielded approximately 230 records, which were screened for relevance based on titles and abstracts, followed by full-text assessment where necessary, resulting in the final corpus of 84 publications.

Inclusion required explicit use of linguistic terminology in relation to programming-related material in the title, abstract, keywords, or main text. The corpus is not intended to be exhaustive, but to provide a representative set of studies in which such designations recur with sufficient regularity to support analytical generalisation.

## 2.3. Analytical representation

Each study is represented through a set of analytical variables corresponding to the aspects of programming-related material specified for analysis. The variable  $O$  denotes the research object addressed in a study (e.g., source code, developer interaction, natural-language artefacts, programming communities). The variable  $A$  denotes analytical focus, i.e., the aspect of the object foregrounded in the study (e.g., naming, readability, documentation, discussion, collaboration). The variable  $T$  denotes terminological designations, i.e. the linguistic terms explicitly invoked (e.g. *text*, *communication*, *discourse*, *genre*, *sublanguage*). The variable  $L$  specifies linguistic status, distinguishing between descriptive use of a term in programming research and its explicit elaboration in relation to established units and relations in linguistic inquiry. The variable  $R$  indicates the role of a study in the analytical procedure, including its contribution to construct identification, grouping, or detailed elaboration.

Each study  $s_i$  is represented as a tuple

$$M(s_i) = (O_i, A_i, T_i, L_i, R_i),$$

which ensures comparability across studies and allows for systematic identification of recurring constructs.

## 2.4. Analytical procedure

Linguistic constructs are defined as elements of a set  $C$ , where each construct corresponds to a terminological designation recorded under  $T$ . Terminological designations are treated as indicators of constructs, which are subsequently specified analytically rather than taken at face value. Their distribution in the corpus is established by registering their presence in individual studies, which permits delineation of the set of constructs relevant for further analysis. The constructs are then grouped according to the objects to which they refer in programming environments.

Analytical reconstruction consists in specifying the constructs in terms of units and relations recognised in linguistic research. This involves placing them alongside established units and relations in the study of text, lexicon, syntax, discourse, and genre, and considering how programming material may be addressed in those terms.

The procedure does not aim at quantitative generalisation or exhaustive coverage. It is directed toward the formulation of theoretically grounded lines of inquiry. The outcome is a set of analytically specified directions for

linguistic investigation, intended as a basis for subsequent empirical work rather than as empirical findings in themselves.

### 3. SPECIFICATION OF THE CORPUS

Research addressing programming practice employs a range of terms when referring to activities and artefacts involved in software development. References to text, communication, documentation, community practice, and the circulation of programming artefacts recur throughout studies of programming environments and collaborative software production. These terms designate observable aspects of programming activity as they appear in written materials and recorded developer interaction. The following subsections organise these references according to their object of reference, distinguishing discussions of source code as readable artefact, natural-language records accompanying software systems, developer interaction, and collective programming practice governing the production, circulation, and reuse of programming artefacts.

#### 3.1. Code as readable artefact

Leaving behind the purely technical analytical perspectives reported in software development research, the first line of inquiry relevant to the present study concerns scholarly acknowledgement of source code as a *readable artefact* encountered by developers during programming practice. This understanding is present in earlier discussions of programme comprehension and literate programming (Knuth, 1984) and is articulated as a distinct line of inquiry in Marino's (2006) account of Critical Code Studies, where he argues that software cannot be understood solely through computational execution, since source code also exists as written material that programmers read while inspecting programme organisation and implementation decisions. Programming languages therefore generate written records that developers process while working with software systems.

This premise altered how software artefacts were addressed in software studies and digital media scholarship, with source code now approached as a readable product as well as a sequence of executable instructions (Dantas et al., 2023; Fuller, 2008; Oliveira et al., 2020). From this standpoint, elements of source code — such as identifiers, literals, and strings — are recognised as meaningful units whose significance to developers arises specifically through the act of reading programme files. This emphasis situates reading alongside computational execution as a condition under which software artefacts acquire meaning.

The status of source code as readable artefact is subsequently tied to properties of the written form, with related discussions in software studies extending the premise of code readability to written expression organised through *textual composition* (Berry, 2011; Marino, 2020;

Montfort et al., 2012). From this standpoint, source code is construed as written artefact whose internal organisation can be examined in terms of structural relations observable in the written form of programme statements.

The textual orientation is developed in scholarship specifying the properties through which textual organisation becomes observable. These studies maintain that programming languages exhibit regular *syntactic organisation* that constrains the form of programme statements and their permissible arrangement in code (Berry, 2011; Montfort et al., 2012; Scott & Aldrich, 2025), adhere to rules of *lexical selection* in the naming of variables, functions, and identifiers (Allamanis et al., 2015; Deißeböck & Pizka, 2006; Newman et al., 2017), and follow *stylistic conventions*, including formatting practices, such as indentation, that visually mark the hierarchical organisation of programme logic (Brock, 2016; Knuth, 1984). In this line of argument, syntactic organisation, lexical selection, and stylistic conventions find written expression in programme statements as sequences legible to developers familiar with the language.

Empirical analyses of software corpora reinforce this designation of programming languages as ordered lexical systems. Studies of identifier naming show that variable and method names follow conventions of semantic transparency, descriptive adequacy, and internal consistency within codebases (Deißeböck & Pizka, 2006; Newman et al., 2017). Investigations of identifier vocabularies further demonstrate stable semantic relations among naming conventions and recurrent identifier names attested across large collections of software projects (Allamanis et al., 2015; Sridhara et al., 2008). This suggests that programming languages maintain organised lexical inventories through which programme components are designated and related to one another by means of semantic relations, a property that invites comparison with natural language at the level of vocabulary. Some authors, however, caution against collapsing programming languages into natural language categories. Cayley (2002), for example, argues that programming languages require distinct reading strategies precisely *because* they possess their own vocabularies and syntactic forms defined by language-specific conventions, and hence should not be equated with natural languages. This position, while rejecting equivalence with natural language, nevertheless recognises source code as a written rule-governed system whose syntax and vocabulary follow conventions specified by the programming language itself.

A related strand of research appraises programming languages as *systems of signs*, with identifiers, reserved keywords, symbolic operators, and punctuation marks constituting sign units, whose meaning is assigned through the rule system of the programming language and is accessed by developers in the course of reading source code

*‘Cayley (2002), for example, argues that programming languages require distinct reading strategies precisely because they possess their own vocabularies and syntactic forms defined by language-specific conventions, and hence should not be equated with natural languages. This position, while rejecting equivalence with natural language, nevertheless recognises source code as a written rule-governed system whose syntax and vocabulary follow conventions specified by the programming language itself.’*

(Deißenböck & Pizka, 2006). These sign units participate in rule-governed combinations that specify procedural relations among instructions and determine the formal organisation of programme statements (Newman et al., 2017). Related work that surveys lexical similarity in software corpora identifies stable semantic relations among identifiers and associated vocabulary, indicating that programming languages contain lexical fields specific to software development (Sridhara et al., 2008). These works justify the view that source code exhibits organised lexical and structural properties characteristic of symbolic systems, meaning that programming languages may be characterised as systems of written signs in which tokens acquire meaning through their position in syntactic configurations and through their lexical specification.

The line of inquiry discussed here characterises source code as a written artefact exhibiting identifiable lexical, syntactic, and stylistic features that together form a system of sign units accessible to developers in the course of reading. In this sense, source code is construed as written executable instructions that permit human comprehension through reading.

### 3.2. Natural-language records

Programming environments contain a wide range of natural-language materials that accompany programme files during software development. Research on programming practice addresses these materials as documents produced alongside executable code (Forward & Lethbridge, 2002; Kalliamvakou et al., 2014). Studies of software projects refer to comments in programme files, repository documentation, README files, bug reports, and issue-tracking records as written explanations of programme behaviour, development decisions, and technical problems encountered during the creation and maintenance of software systems (Bettenburg et al., 2008; Sridhara et al., 2010). These artefacts are a part of the textual record accompanying source code during software construction and revision.

Comments in programme files receive particularly sustained analytical attention. Research on software documentation analyses comment corpora to identify recurrent patterns of written expression present in explanatory annotations accompanying programme statements and serving different explanatory purposes, including descriptions of programme behaviour, statements of implementation assumptions, and warnings concerning limitations or constraints affecting programme execution (Pascarella et al., 2019; Steidl et al., 2013; Tan et al., 2007). Comment analysis also produces taxonomies of annotation types and classification schemes identifying functional categories of comments appearing in source files (Pascarella et al., 2019; Steidl et al., 2013; Yang et al., 2019).

Automated documentation research also addresses these natural-language records. Studies concerned with documentation recovery analyse comment sentences and related textual annotations to extract information concerning programme behaviour, implementation constraints, and design decisions (Panichella et al., 2012; Tan et al., 2007). Computational analysis of documentation artefacts construes these materials as identifiable textual units whose internal structure supports classification and extraction procedures.

Investigations of comment corpora further address the linguistic properties of comment language itself. Some research on software documentation characterises comment language as a specialised *sublanguage* associated with programming practice (Etzkorn et al., 2001; Peck & Brown, 2024; Vinz & Etzkorn, 2008). Studies in this area identify recurrent grammatical structures, condensed syntactic constructions, and domain-specific vocabulary that distinguish it from general-purpose written English (Etzkorn et al., 2001; Vinz & Etzkorn, 2008). In these accounts, comment language is said to display linguistic features dictated by the technical conditions under which developers record explanations of programme behaviour and implementation decisions, which, in their account, warrants its categorisation as a sublanguage.

Repository documentation provides another major category of textual artefact accompanying programme files. README documents present written explanations introducing software projects and providing instructions concerning installation, configuration, and programme usage. Studies of software repositories address README files and related documentation as written materials documenting the purpose and organisation of software systems and providing guidance for readers who encounter programme artefacts in development repositories (Forward & Lethbridge, 2002; Panichella et al., 2012; Yang et al., 2019).

Bug reports and issue-tracking records likewise constitute written documentation in software development. Research analysing issue trackers addresses bug reports

and issue descriptions as textual records documenting software faults, feature requests, and technical discussions of programme maintenance (Ersan, 2023; Panichella et al., 2012). These records contain narrative accounts of software behaviour, descriptions of malfunctioning programme elements, textual explanations accompanying proposed revisions, and requests for additional details required for issue resolution. Studies dealing with issue-tracking archives treat these materials as written documentation preserving records of technical problems and associated solutions (Barua et al., 2014; Rahman et al., 2015).

Mailing-list archives in software projects also contain documentation materials accompanying programme development. Technical correspondence preserved in mailing lists records design discussions, explanations of implementation choices, and project decisions connected to software construction. Research analysing developer mailing lists addresses these records as written materials documenting the evolution of software systems and the reasoning underlying programme modifications (McDaniel & Daer, 2016; Spinuzzi, 2001, 2003).

A related strand of research concerns the designation of natural-language records as *genres* in studies of technical communication and workplace writing. In this literature, artefacts accompanying software development activity are occasionally referred to as *genres of developer communication*. One of the earliest extended discussions appears in the works of Spinuzzi (2001, 2003), whose workplace studies of software development introduce the notion of *genre ecologies* to account for the multiplicity of textual artefacts participating in programming activity.

Spinuzzi's (2001, 2003) analysis relies on ethnographic observation of software developers working on large industrial codebases together with interviews and artefact analysis. The studies record a wide range of written materials that developers consult while working with programme systems. These materials include source code, comments in code files, code examples, technical manuals, search scripts, printed documentation, and electronic communication, with subsequent analysis suggesting that developers move repeatedly among these artefact types while carrying out programming tasks such as locating existing implementations, understanding programme behaviour, or coordinating development work with colleagues. Notable in this enumeration is Spinuzzi's (2001, 2003) inclusion of source code among the recurrent textual artefacts constituting the genre ecology of software development, which means that code itself is viewed as one of the recognised forms of workplace writing mediating programming activity (Spinuzzi, 2001, 2003).

The notion of genre ecology, in Spinuzzi's (2001, 2003) terms, denotes the repertoire of textual artefacts that circulate in software development environments and that developers recognise as familiar forms of technical

communication. In this account, the relation among artefacts receives particular attention. Developers consult code examples while reading programme files, read comments during code inspection, consult manuals when encountering unfamiliar libraries, and exchange messages with collaborators when discussing programme modifications.

Spinuzzi (2001, 2003) also reports variation in the use of comments across development sites. At some locations developers regularly write and maintain multiline comments that record explanations of programme behaviour or future development plans. At other sites, developers treat comments primarily as markers that indicate the location of code sections, and earlier comments sometimes receive little attention because they have not been maintained during subsequent revisions (Spinuzzi, 2003). This suggests that artefact types such as comments participate in established communicative practices that differ from one programming community to another.

Subsequent research on collaborative programming environments adopts related terminology when discussing written interaction in software repositories. Studies of open-source collaboration refer to natural-language records such as issue reports, pull request discussions, and repository documentation as recurring genres of developer communication that organise technical discussion among contributors (McDaniel & Daer, 2016). In this work, the term *genre* designates recognisable forms of written interaction through which participants report software defects, propose revisions, explain implementation decisions, and coordinate collaborative work in distributed programming communities.

Comparable terminology appears in studies addressing documentation practices in software projects. Research on software repositories refers to README files as textual records introducing projects and providing instructions for installation and programme usage (Forward & Lethbridge, 2002). Studies of issue tracking systems analyse bug reports and issue descriptions as recurrent written forms used to record software defects and feature requests (Bettenburg et al., 2008). Investigations of collaborative repositories also address commit messages as short technical texts accompanying revisions to programme files and documenting the reasons for code changes (Tian et al., 2022). In these studies, artefacts related to documentation and project maintenance receive occasional designation as genres of programming practice.

Throughout these accounts, comments, repository documentation, bug reports, issue-tracking records, README files, etc. are referred to as documents produced alongside programme files and consulted during software development. These natural-language records accompanying source code appear under a range of designations, most notably as *sublanguage* and *genre*, with varying degrees of theoretical elaboration.

### 3.3. Developer interaction

Research addressing programming practice presents software development as activity sustained through written interaction among developers working with shared programme artefacts. Studies examining developer collaboration repeatedly refer to exchanges conducted through programme files, repository discussions, and technical correspondence as forms of *communication* among participants engaged in software construction. In this literature, programming environments are posited as settings in which written contributions circulate among developers who read, comment on, and respond to one another's work while maintaining or modifying shared systems.

One strand of research examines large collections of developer discussions hosted on programming forums. Analyses of Stack Overflow archives treat question—answer exchanges as records of technical discussion in which programmers request assistance, offer explanations of programming problems, and exchange programming knowledge with other participants (Barua et al., 2014; Treude et al., 2011). Studies applying topic modelling and classification procedures to these archives characterise the resulting output as record of communication among programmers concerning technical issues encountered during software development (Ferguson et al., 2022).

Comparable descriptions appear in research that deals with written interaction in collaborative repositories. Investigations of GitHub discussions analyse messages appearing in issue trackers, pull-request conversations, and repository comments as written interaction accompanying software development activity (Han et al., 2024). These materials contain technical proposals, requests for clarification, explanations of design decisions, and commentary on proposed code modifications. Studies analysing such records construe repository discussions as evidence of interaction among contributors engaged in collaborative programming work (Storey et al., 2017; Tsay et al., 2014).

Research on programming communication also addresses earlier technical infrastructures such as developer mailing lists and discussion archives. Studies of software projects analyse mailing-list correspondence as written exchanges in which programmers discuss design decisions, report software defects, debate implementation alternatives, propose revisions to existing code, clarify programming conventions, review contributions submitted by other developers, and coordinate development tasks (McDaniel & Daer, 2016; Spinuzzi, 2003). These materials preserve extended threads of developer correspondence in which participants respond to earlier messages and advance collective discussion of technical questions having to do with programme construction and maintenance.

Another body of research refers to developer interaction using the term *discourse* when addressing programming practice or interaction in collaborative development

(Berry, 2004; Brock, 2016, 2019; Brock & Mehlenbacher, 2018; Ersan, 2023; Levi-Eshkol & Ribak, 2025; Marino, 2020; McDaniel & Daer, 2016). Brock (2016, 2019) and Brock and Mehlenbacher (2018) refer to developer discourse in discussions of programming culture and technical communication associated with software production. Marino (2020) likewise refers to discourse when addressing the circulation of programme text and developer commentary in programming environments. In these studies, the term *discourse* serves as a descriptive label for developer interaction or discussion accompanying software production.

The designation refers to written exchanges surrounding programme development, including discussions of implementation strategies, commentary on code fragments, explanations of programme behaviour, proposals for modification, and evaluations of alternative technical solutions. The term also appears in reference to exchanges occurring in mailing lists, issue trackers, programming forums, and repository discussions where developers present problems, respond to technical questions, and negotiate revisions to shared code. In this usage, the term *discourse*, like *communication*, refers to the same body of written exchanges produced in the course of collaborative programming activity associated with the ongoing development, evaluation, and maintenance of software systems.

Professional evaluation practices also treat *source code as written communication addressed to other developers*. Studies examining programming work refer to code review and technical hiring procedures as situations in which developers inspect programme files produced by others and assess programming competence through the written organisation of source code (Hill & Mallette, 2025; Steinert et al., 2010). Source code in these situations becomes material examined by other programmers who reconstruct implementation decisions and technical reasoning from the organisation of programme statements and accompanying comments.

Research on programming practice therefore portrays software development as activity sustained through ongoing interaction among developers. Empirical investigations of this interaction rely on written records preserved in technical infrastructures such as repositories, mailing lists, and programming forums. These exchanges document the processes through which programming tasks are discussed, negotiated, and carried forward during collaborative software production.

In this literature, developer activity surrounding software construction receives repeated characterisation through terms associated with human interaction. References to communication and discourse appear in accounts of programming practice as designations for the collegial exchanges that accompany programme development in shared technical environments.

*‘Importantly, programme artefacts do not remain confined to their initial site of production, appearing in repositories, technical publications, conference demonstrations, educational materials, and media discussions, where fragments of code are encountered as written records open to inspection and commentary. Accounts of software circulation document instances in which code attracts attention far outside development environments, including journalism, blogging, and public debate. Such movement brings source code and related documentation to readers who did not participate in their initial development.’*

### 3.4. Collective programming practice

Research on programming practice presents software development as collective activity in which contributors engage in common technical tasks and participate in the production and maintenance of shared systems (Coleman, 2013; Ensmenger, 2010; Marino, 2006, 2020). Programme files produced by one developer are subsequently read, modified, and extended by others, establishing a continuous chain of contributions in which individual work becomes part of an ongoing technical process.

Participation in such activity presupposes familiarity with conventions that set the terms for the preparation and evaluation of programming artefacts. Expectations concerning readability, stylistic clarity, efficiency, and adherence to established programming conventions define recognised norms of programming practice (Antelmi et al., 2023; Coleman, 2013; Ensmenger, 2010; Kelty, 2008; Mawer, 2025). These norms establish criteria for assessing programme statements, naming practices, and stylistic decisions (Marino, 2020) and are reflected in recurrent features of programming artefacts that appear in successive contributions and programme versions (Coleman, 2013; Dworatzky et al., 2024; Kelty, 2008). The recurrence of these features supports continuity in projects involving multiple contributors and enables subsequent developers to read and modify existing code.

Collaborative development environments provide the material setting in which these practices are realised. Contributors engage with existing codebases, introduce modifications, and extend programme functionality through successive revisions. Work on shared repositories proceeds through contributions that incorporate earlier implementations and add new programme statements and functions, resulting in cumulative programme development maintained through common programming conventions (McDaniel & Daer, 2016; Selic, 2008; Spinuzzi, 2001, 2003).

Participation in such environments leaves persistent records. Development repositories preserve contributor identities, project membership, and licensing arrangements that associate programme artefacts with identifiable groups of developers. Contributor lists, commit histories, and project documentation record successive contributions and connect programme files to the individuals responsible for their creation and modification (El Hafi et al., 2022; Howison & Crowston, 2014; Khan et al., 2022; Marino, 2020).

Importantly, programme artefacts do not remain confined to their initial site of production, appearing in repositories, technical publications, conference demonstrations, educational materials, and media discussions, where fragments of code are encountered as written records open to inspection and commentary (Marino, 2006, 2020). Accounts of software circulation document instances in which code attracts attention far outside development environments, including journalism, blogging, and public debate (Gordon, 2024; Levi-Eshkol & Ribak, 2025). Such movement brings source code and related documentation to readers who did not participate in their initial development. Journalists, policy analysts, educators, and technical commentators encounter fragments of source code in investigative reporting, instructional materials, and institutional discussion, where programme statements serve as documentary evidence, technical illustration, or instructional example. These encounters generate distinct readings of programming artefacts under conditions that involve audiences with differing levels of programming expertise (Kirschenbaum, 2016; Marino, 2020; Navas, 2012; Sack, 2025).

Software development also proceeds through repeated reuse of existing programme material. Developers incorporate earlier implementations into new projects, adapt fragments of code, and produce derivative versions that retain elements of prior contributions. Public repositories support this activity, providing access to programme files for inspection, modification, and redistribution (Coleman, 2013; Gordon, 2024; Han et al., 2024; Marino, 2020). Version control systems preserve records of these processes via documenting successive revisions introduced by multiple contributors. Archived programme versions retain earlier forms of source code and record how later versions incorporate preceding implementations. Programme files therefore carry forward traces of prior contributions that remain visible in subsequent versions.

Furthermore, open-source development environments distribute programme files from one project to another and sustain the reuse of existing implementations. Code fragments and libraries enter new systems as derivative implementations that retain segments of earlier code and introduce further modification (Coleman, 2013; Gordon, 2024). Programme files appear in multiple software systems, each incorporating elements of earlier work.

Programme artefacts also circulate in creative and experimental programming practices. Developers release source code together with commentary that invites modification and reuse in subsequent projects, leading to the production of further variants that incorporate earlier material (Montfort et al., 2012; Navas, 2012; Wolf, 2019).

As a result of these processes, source code and related documentation persist as a cumulative record of contributions produced, read, and modified by multiple participants over time. Programme artefacts remain associated with identifiable contributors while continuing to appear in new settings in which they are encountered, evaluated, and reused.

The lines of research discussed here use a set of recurring terms to refer to the collective and professional aspects of programming practice. References to *programming (or developer) communities* appear in studies addressing collaborative development and participation in common technical work (Antelmi et al., 2023; Coleman, 2013; Dworatzky et al., 2024; Ensmenger, 2010). The term *programming culture* is used in accounts of conventions, vocabularies, and stylistic expectations that regulate code production and evaluation (Kelty, 2008; Marino, 2020; Selic, 2008). Discussions of contributor records, attribution, and continued participation in software projects introduce the notion of

*professional identity* in relation to developers whose contributions remain visible in programme artefacts and their revision histories (El Hafi et al., 2022; Howison & Crowston, 2014; Khan et al., 2022; Mawer, 2025; Wolf, 2019).

These terms appear in the literature as established labels referring to collective participation, normative expectations, and the attribution of contributions in software development.

#### 4. RESULTS AND DISCUSSION

##### 4.1. Categorisation and inventory of linguistic constructs in programming scholarship

The present stage of analysis proceeds from a corpus of studies assembled through systematic mapping and analytical classification, in which linguistic constructs are repeatedly referenced in relation to programming environments. A linguistic construct is defined here as *an analytical unit through which programming scholarship designates linguistic phenomena in software development*.

The recorded constructs were grouped according to the objects to which they refer in programming environments, resulting in four groups of constructs: (i) code as readable artefact; (ii) natural-language records; (iii) developer interaction; and (iv) collective programming practice (Figure 1).

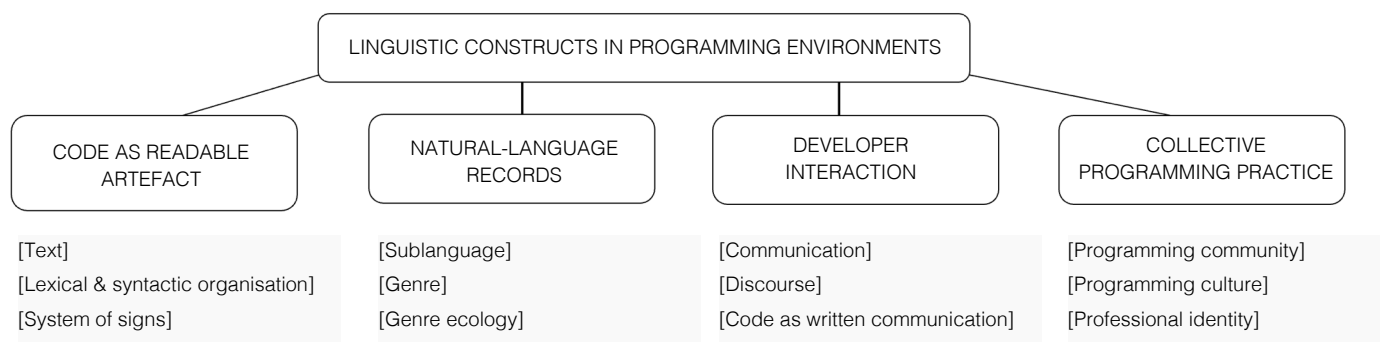


Figure 1. Inventory of linguistic constructs in programming environments

The grouping of linguistic constructs follows from differences in the referential scope assigned to them in programming scholarship. Each group corresponds to a distinct class of written or recorded material in programming environments to which linguistic designations are applied.

The first group concerns source code as a readable artefact. References to *text*, *lexical selection*, *syntactic organisation*, *stylistic conventions*, and *systems of signs* point to properties of code as written material. These designations correspond to units and relations present in programme files and accessible to developers through reading.

The second group concerns natural-language records accompanying source code. References to *sublanguage* and *genre* are registered in discussions of materials produced

alongside programme files, including comments, repository documentation, issue reports, and related artefacts that record explanations, instructions, and technical descriptions connected to software development.

The third group concerns developer interaction. References to *communication* and *discourse* refer to written exchanges among developers, including forum discussions, repository conversations, and mailing-list correspondence. These designations correspond to sequences of contributions in which participants address one another in the course of software development.

The fourth group concerns collective programming practice. References to *programming community*, *programming culture*, and *professional identity* are registered in

discussions of collective participation, normative expectations, and attribution of contributions. These designations correspond to the organisation of programming activity as a socially sustained form of work.

The classification proceeds by grouping linguistic designations according to the objects to which they refer, distinguishing code, accompanying natural-language records, developer interaction, and collective participation as analytically distinct referents in programming environments.

#### 4.2. Text and sign

Across the material reviewed, source code is consistently positioned as a readable artefact through the recurrent referral to code as text, the invocation of its lexical and syntactic organisation, the attribution of stylistic properties, and the characterisation of code as a system of signs.

In the studies reviewed, the construct *text* appears as a designation that places source code among readable artefacts and relates it to written material. Cayley (2002) refers to code as '*text read as language displayed on a screen*'. Marino (2006) presents code as a '*cultural text worthy of analysis and rich with possibilities for interpretation*'. Berry (2011) reinforces this designation through comparison with written artefacts such as musical scores and architectural plans. Fuller (2008) refers to code as a textual artefact within a practice of writing. Across the remaining corpus, the construct *text* similarly functions as a classificatory label establishing readability and supporting comparison with other written forms.

While these accounts tend to equate code with text, the criteria developed in linguistic scholarship for establishing code as text are not brought into the analysis. This leaves open the question of how such criteria might be brought into the analysis of code.

One way of pursuing this is through text linguistics, particularly the systemic-functional account of Halliday and Hasan (2014), where text is defined as a semantic unit characterised by cohesion, coherence, and completion. From this point of view, one might look more closely at the relations linking successive segments, including causality, temporality, and purpose, as well as the means by which continuity is maintained, such as lexical repetition, naming consistency, and reference tracking in identifiers and comments.

The designation of code as text also raises the question of segmentation into text-forming units. Here, Blokh's (1986) notion of the dicteme introduces criteria for identifying minimal units based on thematic unity and communicative completeness, which makes it possible to consider whether sequences of statements form bounded semantic wholes or remain only formally adjacent. This brings into play a distinction between sequences that hold together semantically and those that simply follow one another in linear order.

Further specification may address the relation between formally discrete segments and their semantic connectedness, including the extent to which ordering corresponds to progression of states, conditions, or procedures. This question may also be pursued in the study of information structure (Firbas, 1992), where ordering is analysed in terms of the distribution of given and new information and the progression of thematic development. It is also worth considering how links between segments are made explicit or left implicit, and how much must be recovered from naming or position alone.

This opens the possibility of considering the status of code as text on the basis of explicit linguistic criteria.

At the same time, the designation of code as text brings into view its internal organisation, including the treatment of lexical units. In the corpus studied, lexical structure is introduced through only partially compatible definitions. Scott and Aldrich (2025) define a programme as a sequence of tokens and tokens as the shortest strings of characters with individual meaning, restricting lexicality to formally generated units, whereas Oliveira et al. (2020) identify naming conventions as a factor affecting code readability and refer to identifier naming in terms that presuppose word segmentation, which permits considering identifiers in terms of their internal lexical structure and informational contribution. Newman et al. (2017) recast this perspective in programme-internal terms, proposing a set of lexical categories for source-code identifiers determined by syntactic behaviour rather than by natural-language semantics, even though references to words and naming practices remain present.

Lexicality is thus introduced through tokens, word units, and naming practices, without explicit specification of the linguistic criteria by which lexical units are defined, leaving these designations available for further articulation in linguistic terms.

Here, work in lexicology offers a more precise point of departure, where the lexeme is defined as an abstract unit underlying word forms (Cruse, 1986; Lyons, 1977), and where distinctions between lexical and grammatical meaning, as well as internal morphological structure, are explicitly defined. From this angle, identifiers can be considered as composite lexical formations, including compounding, abbreviation, and truncation, while multi-word identifiers invite comparison with phraseological units. One may also consider how naming practices relate to paradigmatic relations such as synonymy or hyponymy, and to syntagmatic constraints on co-occurrence and ordering. This opens the possibility of asking whether identifiers form a restricted lexical set with recurrent relations, or a collection of ad hoc labels.

In the corpus of studies reviewed, syntactic organisation is likewise introduced through heterogeneous formulations that refer to structure, context, and representation.

*‘From this angle, identifiers can be considered as composite lexical formations, including compounding, abbreviation, and truncation, while multi-word identifiers invite comparison with phraseological units. One may also consider how naming practices relate to paradigmatic relations such as synonymy or hyponymy, and to syntagmatic constraints on co-occurrence and ordering. This opens the possibility of asking whether identifiers form a restricted lexical set with recurrent relations, or a collection of ad hoc labels.’*

Allamanis (2015) refers to ‘*syntactic structure of the code*’ (Allamanis, 2015, p. 47) and ‘*syntactic context*’ (Allamanis, 2015, p. 41) within statistical modelling, Marino (2020) describes formal syntax as defining valid statements and expressions and links it to sequential and hierarchical arrangement, and Deißeböck and Pizka (2006, p. 264) view identifiers as ‘*syntactic entities*’ within representational structures such as the abstract syntax tree. Here, syntax appears in connection with well-formedness, structural arrangement, and representation, with terminology that remains tied to programming descriptions and does not converge on a shared set of linguistic categories.

Linguistic work on syntax offers a different level of precision, for example in Functional Grammar (Dik & Hengeveld, 1997) or Role and Reference Grammar (Van Valin, 2014), where attention is given to predicate–argument structure, grammatical relations, and valency. In this light, code segments may be examined in terms of constituent relations, dependency, and argument realisation, alongside distinctions such as subcategorisation and transitivity. It also becomes possible to ask how heads and dependents are organised, how sequences are built, and whether relations resemble endocentric or exocentric constructions. Questions of alignment, adjacency, and discontinuity can also be reconsidered in linguistic terms.

References to style, in turn, appear in connection with readability, presentation, and coding conventions, with Fuller (2008) invoking evaluative descriptions of code as ‘*beautiful*’ (Fuller, 2008, p. 117) and ‘*elegant*’ (Fuller, 2008, p. 43), Scott and Aldrich (2025) emphasising that decisions of style are paramount in regards to making a readable programme, and Marino (2020) specifying readability through features such as syntax, phrasing, indentation, white space, conventions, and overall organisation. These references connect textual form and readability, with stylistic characterisation expressed in practical and evaluative terms, without recourse to clearly delimited linguistic categories.

Work in stylistics may provide a basis for such analysis, particularly in functional stylistics and systemic-functional linguistics (Bell, 2002; Eggins, 2004), where style is treated as variation in linguistic choice. From this angle, one may look at recurring features such as lexical selection, degrees of syntactic complexity, and forms of compression or expansion. Naming practices can be considered in terms of consistency and variation, while abbreviated and expanded forms raise questions about economy and explicitness. Readability, in turn, may be linked to formal properties of the text rather than to general impressions.

Finally, references to code elements as signs within a system appear across the material with varying scope and emphasis. In Scott and Aldrich (2025), the term *sign* refers to symbolic elements such as *sign bit*, *plus sign (+)*, and *the initial hash sign*, remaining tied to formal notation. In Marino (2020), signs are explicitly connected with meaning, as in ‘*signs have meaning through their relation to the full language*’ (Marino, 2020, p. 236) and ‘*each sign in one system can be equated to a sign in another system*’ (Marino, 2020, p. 42), with the relation between sign and meaning noted without further specification in semiotic terms.

Semiotic theory may be able to provide a clearer set of distinctions here, including De Saussure’s (1916) relation between signifier and signified and Peirce’s (1985) triadic account involving the interpretant. These allow one to ask how elements of code relate to meaning, reference, and interpretation, and how formal representation connects with what is understood. One may also consider relations among signs, including syntagmatic and paradigmatic organisation (Jakobson, 1960), and the way larger textual formations may be viewed as complex signs. Questions of syntax, semantics, and pragmatics, together with semiosis and interpretation effects, follow from this line of inquiry.

#### 4.3. Sublanguage and genre

In the corpus studied, natural-language material accompanying software is consistently discussed with reference to *sublanguage* and *genre*, applied to artefacts such as comments, documentation, repository records, and related textual materials produced in the course of development. These constructs are introduced under different criteria, including communicative purpose, grammatical properties, recurrence, and participation in development activity, and are specified to varying degrees, from explicit definition and structural characterisation to classificatory or descriptive use.

The construct *sublanguage* is introduced as a domain-restricted subset of natural language in Etzkorn et al. (2001) and Vinz and Etzkorn (2008), where the focus is on English-language comments. Identification rests on grammatical properties such as sentence structure, tense, and telegraphic compression, which give rise to reduced syntactic realisation and a restricted semantic range.

*‘Attention may also be given to the relation between syntax and semantics in the subset, in particular to the extent to which semantic distinctions are reflected in grammatical subclasses. In sublanguage theory, semantic roles and relations are often tied to specific syntactic configurations, yielding tightly coupled form–meaning correspondences. This raises the question of whether programming-related natural-language records exhibit comparable correspondences between types of processes and their grammatical realisation.’*

In Peck and Brown (2024), the range is widened to include what are termed ‘source code annotations’, covering comments, commit messages, and documentation. The construct is applied to a heterogeneous set of artefacts characterised in terms of written modality, relation to code, and extractability as corpus data, including the presence of embedded code fragments. Criteria of identification combine grammatical features with explanatory purpose and procedures of computational processing. The construct retains its definition as a restricted variety, while the range of materials and criteria becomes more extensive.

In these studies, sublanguage is introduced as a restricted variety of natural language, first identified through grammatical features of comments and later extended to a wider set of textual artefacts associated with programming activity. This extension increases the empirical range while leaving open the question of how the subset may be delimited in linguistic terms.

In linguistic theory, sublanguage is not established through the presence of reduced or specialised features alone, but through the identification of a subset defined by formal and distributional constraints. In the work of Harris (1989), this involves determining whether the sentences of a given corpus form a system closed under specific grammatical operations. One line of inquiry would therefore consist in testing whether combinations of constructions found in programming-related records remain admissible under coordination, embedding, or extension, or whether such operations introduce forms that fall outside the subset.

A further step concerns the internal organisation of grammatical classes. In sublanguage analysis, subclasses of nouns, verbs, and predicates are defined through co-occurrence restrictions, such that only certain combinations are permitted. This makes it possible to determine whether elements such as identifiers, actions, and states participate in stable distributional classes, or whether their combinatory behaviour remains unrestricted.

Another point concerns the identification of canonical sentence types, which in Harris’s (1968, 1988) account serve as the basis for describing the grammar of the subset. This involves determining whether a limited set of recurrent constructions can be established that account for the majority of observed sentences, and whether more complex or infrequent forms can be derived from these through systematic transformations.

Attention may also be given to the relation between syntax and semantics in the subset, in particular to the extent to which semantic distinctions are reflected in grammatical subclasses. In sublanguage theory, semantic roles and relations are often tied to specific syntactic configurations, yielding tightly coupled form–meaning correspondences. This raises the question of whether programming-related natural-language records exhibit comparable correspondences between types of processes and their grammatical realisation.

Finally, the status of the subset may be assessed in terms of its internal boundedness. In corpus-based sublanguage research, this is examined through the stabilisation of lexical, syntactic, and combinatory inventories as corpus size increases. This makes it possible to determine whether the material forms a finite system with predictable distributions or remains open-ended.

These criteria define sublanguage as a formally delimited subset characterised by closure, restricted co-occurrence, canonical constructions, and systematic relations between form and meaning, providing a basis for assessing whether programming-related natural-language material satisfies the conditions established in linguistic theory.

Next up, the construct *genre* appears in the corpus with different degrees of specification, ranging from a loosely applied classificatory label attached to groups of artefacts (Rahman et al., 2018) to more explicit formulations grounded in recurrence, stability, and relations among textual types (Spinuzzi, 2001). In some cases, the term accompanies empirically delimited sets such as comments or issue discussions without linguistic characterisation (Spinuzzi, 2003), while in others it is defined through repeated use, coordination with other textual forms, or internally organised document types such as use cases (Williams, 2003). Across these uses, identification rests on functional grouping and recurrence, while linguistic criteria remain largely implicit.

In linguistic work, *genre* is established through the recurrence of linguistically identifiable types of utterance, rather than through grouping of artefacts alone. In the work of Bakhtin (2010), genres are defined as relatively stable types of utterances characterised by typical compositional structure, thematic content, and style. This places emphasis on the internal organisation of texts and on recurrent linguistic features that distinguish one genre from another. One line of inquiry would therefore consist in

*‘Another point concerns textual organisation, including the sequencing of segments and the distribution of functions across them. In genre theory, this includes recurrent patterns of opening, development, and closure, as well as conventionalised arrangements of information, as formalised in move-based accounts of genre. This raises the question of whether natural-language records exhibit comparable regularities in the ordering and composition of segments.’*

identifying whether programming artefacts exhibit stable configurations of clause structure, lexical selection, and sequencing that recur across instances of the same textual type. This includes the possibility that particular genres are associated with recurrent syntactic constructions, preferred lexical sets, or fixed patterns of progression.

Further specification may be grounded in corpus-based genre analysis, such as the work of Biber (2010), where genres are differentiated through systematic variation in linguistic features. This makes it possible to test whether groups such as comments, bug reports, or documentation are associated with distinct distributions of grammatical and lexical features, rather than being defined solely by their role in development activity.

Another point concerns textual organisation, including the sequencing of segments and the distribution of functions across them. In genre theory, this includes recurrent patterns of opening, development, and closure, as well as conventionalised arrangements of information, as formalised in move-based accounts of genre (Swales, 1990). This raises the question of whether natural-language records exhibit comparable regularities in the ordering and composition of segments.

Attention may also be given to relations among genres, including cases where texts occur in coordinated sets or sequences. While this has been addressed in programming studies in terms of ‘genre ecologies’ (Spinuzzi, 2003), a linguistic account would consider whether such relations are reflected in recurrent linguistic dependencies or cross-references between textual types. Spinuzzi (2003) also extends the term to source code itself, raising the question of whether formally specified programme text may be addressed in genre terms and, if so, on what linguistic grounds such classification could be sustained.

From this perspective, genre is established through the recurrence of linguistically identifiable features and organisational patterns that distinguish one type of text from another, providing a basis for determining whether natural-language documents form stable genres in the linguistic sense.

#### 4.4. Communication and discourse

In the literature reviewed, developer interaction is referred to through the terms *communication* and *discourse*, both applied to written exchanges accompanying software development in repositories, forums, and mailing lists (Barua et al., 2014; Brock, 2016; Han et al., 2024; Marino, 2020; McDaniel & Daer, 2016; Storey et al., 2017). The same stream of messages carries questions, explanations, proposed changes, and responses exchanged in the course of work on shared code.

The same material is described as communication in studies concerned with interaction, coordination, and knowledge exchange, and as discourse in work addressing programming practice and technical culture (Brock & Mehlenbacher, 2018; Eshkol & Ribak, 2025; Ferguson et al., 2022; Levi-Tsay et al., 2014). The artefacts remain the same, and so do the analytical procedures. The two terms circulate side by side and attach to the same sets of exchanges, without criteria that would separate one from the other in linguistic terms.

In linguistics, the distinction between communication and discourse is drawn explicitly and may prove helpful when applied to studies of developer interaction. Communication is examined as sequences of acts carried out by speakers or writers and is formalised in pragmatics through speech act theory (Austin, 1975; Searle, 1969). An utterance is considered at several levels at once: its locutionary content, its illocutionary force such as requesting, informing, proposing, or evaluating, and its possible perlocutionary effects, for instance prompting a reply, securing agreement, or triggering revision. From this point of view, communication appears as a chain of acts linked to participants, intentions, and responses.

Discourse is addressed at a different level and concerns how stretches of language hold together across turns and contributions. This includes continuity of reference, development of topics, and the ordering of information from one message to the next. It also includes recurrent types of utterance associated with particular settings, described by Bakhtin (1994) as relatively stable forms tied to recurring situations.

Hence, developer interaction can be examined along two lines. One centres on communication as sequences of acts, where messages take the form of questions, answers, requests for clarification, proposals, or evaluations, and where attention is given to how one move prompts another and how participants take up or resist what has been said. The other centres on discourse as extended sequences, where the focus falls on how threads develop, how earlier contributions are reformulated or carried forward, how topics are maintained or brought to a close, and how coherence is sustained over time in multi-participant exchange. This makes it possible to separate communication and discourse as two analytical levels applied to the

same material. Communication refers to individual acts and their immediate effects. Discourse concerns the way these acts accumulate into connected stretches of interaction that display continuity and recognisable organisation.

#### 4.5. Community, culture and identity

Finally, in the corpus studied here, collective aspects of programming practice are referred to through the constructs *programming community*, *programming culture*, and *professional identity*, all used in connection with collaborative development, shared conventions, and the attribution of contributions (Coleman, 2013; El Hafi et al., 2022; Ensmenger, 2010; Howison & Crowston, 2014; Kelty, 2008; Marino, 2020). These terms accompany descriptions of participation in common technical work, the persistence of conventions in code, and the visibility of individual contributors in programme artefacts and their revision histories. The same body of material — programme files, repositories, and records of contribution — is associated with communities when participation and collaboration are in view, with culture when conventions and expectations are discussed, and with identity when attribution and continued involvement are foregrounded. The terms thus follow different descriptive emphases, while leaving open the possibility of examining these phenomena through linguistic criteria.

In sociolinguistics and discourse studies, a programming community may be considered in relation to the notion of a speech or discourse community (Hymes, 1989; Swales, 1990), where a group is characterised by a shared repertoire of lexical items, recurrent expressions, and recognised textual types. This makes it possible to ask whether contributors working on the same projects display consistent lexical selection, stable terminological usage, and recurrent forms of expression that persist across contributions. Cohesion across texts may also be taken into account, following Halliday and Hasan (2014), including repetition of key terms and continuity of reference in naming practices that link individual contributions into a recognisable linguistic whole.

Programming culture may be examined in terms of recurrent ways of formulating evaluation and instruction, drawing on work in stylistics and corpus linguistics (Halliday, 2014; Malyuga, 2024). This includes the presence of conventionalised expressions, formulaic sequences, and recurring evaluative constructions that appear in discussions of code. Metalinguistic statements — such as formulations expressing obligation, recommendation, or prohibition — provide direct linguistic evidence of normative expectations, as they articulate what is considered acceptable or desirable in programming practice.

Professional identity may be approached through the linguistic marking of stance and positioning in interaction, based on work on stance and evaluation (Biber, 2021;

Hunston, 2010), indexicality (Silverstein, 2003), and linguistic identity in sociolinguistics (Bucholtz & Hall, 2005). This includes the use of specialised vocabulary, expressions of certainty or caution, and forms that convey authority or participation in established practices. Reference to self and others, including pronouns and directive constructions, further situates contributors in relation to one another and to the work under discussion through repeated forms of linguistic positioning.

These lines of inquiry relate programming community, culture, and professional identity to identifiable linguistic features that mark participation, evaluation, and positioning in programming interaction.

#### 5. CONCLUSION

The present study has taken as its starting point a simple observation that programming research already makes sustained use of linguistic terminology when addressing code, documentation, interaction, and collective activity. These terms appear with sufficient regularity to suggest that programming environments are already being approached, in practice, as sites where language matters. What has been lacking is not recognition of these phenomena, but their specification in terms that would make them accessible to linguistic inquiry proper.

The analysis has therefore proceeded by placing these designations alongside distinctions long established in linguistics and asking what follows when they are brought to a more explicit level of discussion. This does not amount to a transfer of labels from one field to another, but involves a change in the level at which the material is considered. Questions of readability are recast in terms of textuality, naming practices are situated in relation to lexical structure, and repository discussions are approached as material for the analysis of speech acts and extended discourse. What appears in programming research as community, culture, or identity becomes available as evidence of shared repertoires, evaluative language, and positioning in interaction.

Hence, the contribution of the study is not the introduction of new constructs, but the re-specification of already circulating ones. This is where the principle of division of labour becomes important. Programming research provides access to material, tasks, and environments in which these phenomena are produced and observed. Linguistics, in turn, can offer a set of distinctions, units, and procedures that make it feasible to approach this material through analytical categories established in language research. The two lines of work remain complementary, each retaining its own analytical priorities.

The directions proposed here should be read in this light. They do not claim to exhaust what can be said about programming environments from a linguistic point of view, and they do not assume that the questions raised

have not been addressed in isolated studies. It is entirely possible that particular aspects have already been examined in work that falls outside the present corpus. What appears more tentative at this stage is a systematic account that would bring these strands together and treat programming material as a coherent object of linguistic inquiry.

At the same time, the proposals remain selective. They identify points at which linguistic theory offers clear entry into the material, but they do not attempt to cover all possible levels of analysis or all types of programming artefacts. The study has also been based on a deliberately delimited corpus assembled to reflect recurring terminology, without seeking comprehensive coverage of all relevant research. This inevitably leaves out work that might complicate or refine the picture.

## References

- Abdelsalam, Y., Peitek, N., Bergum, A., & Apel, S. (2026). The effect of comments on program comprehension: An eye-tracking study. *Empirical Software Engineering*, 31(4), Article 94. <https://dx.doi.org/10.1007/s10664-025-10721-2>
- Allamanis, M., Barr, E. T., Bird, C., & Sutton, C. (2015, August). Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (pp. 38–49). ACM. <https://doi.org/10.1145/2786805.2786849>
- Antelmi, A., Cordasco, G., De Vinco, D., & Spagnuolo, C. (2023, April). The age of snippet programming: Toward understanding developer communities in Stack Overflow and Reddit. In *Companion Proceedings of the ACM Web Conference 2023* (pp. 1218–1224). ACM. <https://doi.org/10.1145/3543873.3587673>
- Austin, J. L. (1975). *How to do things with words*. Clarendon Press.
- Bakhtin, M. (1994). *Discourse in life and discourse in art* (Vol. 10). Lawrence Erlbaum.
- Bakhtin, M. M. (2010). *Speech genres and other late essays*. University of Texas Press.
- Barua, A., Thomas, S. W., & Hassan, A. E. (2014). What are developers talking about? An analysis of topics and trends in stack overflow. *Empirical Software Engineering*, 19(3), 619–654. <https://dx.doi.org/10.1007/s10664-012-9231-y>
- Bell, A. (2002). Back in style: Reworking audience design. In P. Eckert & J. R. Rickford (Eds.), *Style and sociolinguistic variation* (pp. 139–169). Cambridge University Press.
- Berry, D. (2004). The contestation of code: A preliminary investigation into the discourse of the free/libre and open source movements. *Critical Discourse Studies*, 1(1), 65–89. <https://doi.org/10.1080/17405900410001674524>
- Berry, D. (2011). *The philosophy of software*. Palgrave Macmillan.
- Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., & Zimmermann, T. (2008, November 9–14). What makes a good bug report? In *Proceedings of the 16th International Symposium on Foundations of Software Engineering (SIGSOFT'08/FSE-16)* (pp. 308–318). ACM. <https://dx.doi.org/10.1145/1453101.1453146>
- Biber, D. (2010). What can a corpus tell us about registers and genres? In A. O'Keeffe & M. McCarthy (Eds.), *The Routledge handbook of corpus linguistics* (pp. 241–254). Routledge. <https://doi.org/10.4324/9780203856949>
- Biber, D. (2021). Stance and grammatical complexity: An unlikely partnership discovered through corpus analysis. In D. Biber, B. Gray, S. Staples, & J. Egbert (Eds.), *The register-functional approach to grammatical complexity: Theoretical foundation, descriptive research findings, application* (pp. 113–128). Routledge. <https://dx.doi.org/10.4324/9781003087991>
- Blokh, M. Y. (1986). *Theoretical foundations of grammar*. Higher School.
- Brock, K. (2016, January 15). The 'FizzBuzz' programming test: A case-based exploration of rhetorical style in code. *Computational Culture*, 5. Retrieved from <http://computationalculture.net/the-fizzbuzz-programming-test-a-case-based-exploration-of-rhetorical-style-in-code>
- Brock, K. (2019). *Rhetorical code studies: Discovering arguments in and around code*. University of Michigan Press.
- Brock, K., & Mehlenbacher, A. R. (2018). Rhetorical genres in code. *Journal of Technical Writing and Communication*, 48(4), 383–411. <https://dx.doi.org/10.1177/0047281617726278>
- Bucholtz, M., & Hall, K. (2005). Identity and interaction: A sociocultural linguistic approach. *Discourse Studies*, 7(4-5), 585–614. <https://dx.doi.org/10.1177/1461445605054407>
- Cayley, J. (2002, September 10). The code is not the text (unless it is the text). *Electronic Book Review*. Retrieved from <https://electronicbookreview.com/publications/the-code-is-not-the-text-unless-it-is-the-text>
- Coleman, G. (2013). *Coding freedom: The ethics and aesthetics of hacking*. Princeton University Press.
- Croft, B., & Lafferty, J. (Eds.). (2003). *Language modeling for information retrieval* (Vol. 13). Springer Science & Business Media. <https://dx.doi.org/10.1007/978-94-017-0171-6>
- Cruse, D. A. (1986). *Lexical semantics*. Cambridge University Press.

- Dantas, C. E. C., Rocha, A. M., & Maia, M. A. (2023, October). How do developers improve code readability? An empirical study of pull requests. In *Proceedings of the 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 110–122). IEEE. <https://doi.org/10.1109/ICSME58846.2023.00022>
- De Freitas Farias, M. A., de Mendonça Neto, M. G., Kalinowski, M., & Spínola, R. O. (2020). Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. *Information and Software Technology, 121*, Article 106270. <https://dx.doi.org/10.1016/j.infsof.2020.106270>
- De Saussure, F. (1916). *Course in general linguistics*. Columbia University Press.
- Deißenböck, F., & Pizka, M. (2006). Concise and consistent naming. *Software Quality Journal, 14*(3), 261–282. <https://doi.org/10.1007/s11219-006-9219-1>
- Dik, S. C., & Hengeveld, K. (Eds.) (1997). *The theory of functional grammar: The Structure of the Clause*. Mouton de Gruyter.
- Dworatzky, K., Dekorsy, V., & Theis, S. (2024, June). Decoding the diversity of the german software developer community: Insights from an exploratory cluster analysis. In *Proceedings of the International Conference on Human-Computer Interaction* (pp. 275–295). Springer. [https://doi.org/10.1007/978-3-031-60125-5\\_19](https://doi.org/10.1007/978-3-031-60125-5_19)
- Eggins, S. (2004). *Introduction to systemic functional linguistics*. A&C Black.
- El Hafi, L., Garcia Ricardez, G. A., von Drigalski, F., Inoue, Y., Yamamoto, M., & Yamamoto, T. (2022). Software development environment for collaborative research workflow in robotic system integration. *Advanced Robotics, 36*(11), 533–547. <https://dx.doi.org/10.1080/01691864.2022.2068353>
- Ensmenger, N. (2010). *The computer boys take over: Computers, programmers, and the politics of technical expertise*. MIT Press.
- Ersan, E. N. (2023). *Beyond the chat: Abstractive summarization and pos analysis of developer discourse on discord*. Carleton University Press.
- Etzkorn, L. H., Davis, C. G., & Bowen, L. L. (2001). The language of comments in computer software: A sublanguage of English. *Journal of Pragmatics, 33*(11), 1731–1756. [https://dx.doi.org/10.1016/S0378-2166\(00\)00068-0](https://dx.doi.org/10.1016/S0378-2166(00)00068-0)
- Ferguson, S. A., Cheng, K., Adolphe, L., Van de Zande, G., Wallace, D., & Olechowski, A. (2022). Communication patterns in engineering enterprise social networks: An exploratory analysis using short text topic modelling. *Design Science, 8*, Article e18. <https://doi.org/10.1017/dsj.2022.12>
- Firbas, J. (1992). *Functional sentence perspective in written and spoken communication*. Cambridge University Press.
- Fluri, B., Wursch, M., & Gall, H. C. (2007, October). Do code and comments co-evolve? On the relation between source code and comment changes. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE 2007)* (pp. 70–79). IEEE. <https://dx.doi.org/10.1109/WCRE.2007.21>
- Fluri, B., Würsch, M., Giger, E., & Gall, H. C. (2009). Analyzing the co-evolution of comments and source code. *Software Quality Journal, 17*, 367–394. <https://dx.doi.org/10.1007/s11219-009-9075-x>
- Forward, A., & Lethbridge, T. C. (2002, November). The relevance of software documentation, tools and technologies: A survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering* (pp. 26–33). ACM. <https://doi.org/10.1145/585064.585065>
- Fuller, M. (Ed.). (2008). *Software studies: A lexicon*. MIT Press.
- Gordon, C. S. (2024, October 17). The linguistics of programming. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (pp. 162–182). ACM. <https://dx.doi.org/10.1145/3689492.3689806>
- Halliday, M. A. K. (2014). Language as social semiotic. In J. Angermüller, D. Maingueneau, & R. Wodak (Eds.), *The discourse studies reader* (pp. 263–272). John Benjamins Publishing Company. <https://dx.doi.org/10.1075/z.18.4.52hal>
- Halliday, M. A. K., & Hasan, R. (2014). *Cohesion in English*. Routledge. <https://doi.org/10.4324/9781315836010>
- Han, Y., Wang, Z., Feng, Y., Zhao, Z., & Wang, Y. (2024). Characterizing developers' linguistic behaviors in open source development across their social statuses. *Proceedings of the ACM on Human-Computer Interaction, 8*, Article 29. <https://doi.org/10.1145/3637306>
- Harris, Z. (1968). *Mathematical structures of language*. Wiley.
- Harris, Z. (1988). *Language and information*. Columbia University Press.
- Harris, Z. (1989). Discourse and sublanguage. In R. Kittredge & J. Lehrberger (Eds.), *Studies of language in restricted semantic domains* (pp. 231–236). Walter de Gruyter.
- Hill, A., & Mallette, J. C. (2025, October 27–31). The complexity of a simple code comment: Genre conventions, social influence, and rhetorical functions. In *Proceedings of the 43rd ACM International Conference on Design of Communication (SIGDOC'25)* (pp. 26–31). ACM. <https://doi.org/10.1145/3711670.3764616>
- Howison, J., & Crowston, K. (2014). Collaboration through open superposition: A theory of the open source way. *MIS Quarterly, 38*(1), 29–50. <https://dx.doi.org/10.25300/MISQ/2014/38.1.02>
- Hunston, S. (2010). *Corpus approaches to evaluation: Phraseology and evaluative language*. Routledge. <https://dx.doi.org/10.4324/9780203841686>
- Hymes, D. H. (1989). Ways of speaking. In D. Bauman & J. Sherzer (Eds.), *Explorations in the ethnography of speaking* (pp. 433–451). Cambridge University Press.
- Iyer, S., Konstas, I., Cheung, A., & Zettlemoyer, L. (2016, August). Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics* (pp. 2073–2083). ACM. <https://dx.doi.org/10.18653/v1/P16-1195>
- Jakobson, R. (1960). Closing statement: Linguistics and poetics. In T. A. Sebeok (Ed.), *Style in language* (pp. 350–377). MIT Press.

- Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., & Damian, D. (2014). *The promises and perils of mining GitHub*. MSR.
- Kelty, C. (2008). *Two bits: The cultural significance of free software*. Duke University Press.
- Khan, M. S., Khan, A. W., Khan, F., Khan, M. A., & Whangbo, T. K. (2022). Critical challenges to adopt DevOps culture in software organizations: A systematic review. *IEEE Access*, 10, 14339–14349. <https://dx.doi.org/10.1109/ACCESS.2022.3145970>
- Kirschenbaum, M. (2016). *Track changes: A literary history of word processing*. Harvard University Press.
- Knuth, D. (1984). Literate programming. *The Computer Journal*, 27(2), 97–111. <https://doi.org/10.1093/comjnl/27.2.97>
- Levi-Eshkol, K., & Ribak, R. (2025). ‘Track every move’: Analyzing developers’ privacy discourse in GitHub README files. *New Media & Society*, 27(12), 6534–6550. <https://doi.org/10.1177/14614448241270541>
- Lyons, J. (1977). *Semantics (Vol. 2)*. Cambridge University Press.
- Malyuga, E. N. (2024). *The language of corporate communication: Functional, pragmatic and cultural dimensions*. Springer. <https://doi.org/10.1007/978-3-031-58905-8>
- Marino, M. C. (2006, December 4). Critical code studies. *Electronic Book Review*. Retrieved from <https://electronic-bookreview.com/publications/critical-code-studies>
- Marino, M. C. (2020). *Critical code studies*. MIT Press.
- Mawer, D. (2025, November). WIP: Exploring programmer identity in the figured world of an undergraduate creative computing classroom. In *Proceedings of the IEEE Frontiers in Education Conference (FIE)* (pp. 1–5). IEEE. <https://doi.org/10.1109/FIE63693.2025.11328577>
- McDaniel, R., & Daer, A. (2016). Developer discourse: Exploring technical communication practices within video game development. *Technical Communication Quarterly*, 25(3), 155–166. <https://dx.doi.org/10.1080/10572252.2016.1180430>
- Montfort, N., Baudoin, P., Bell, J., Bogost, I., Douglass, J., Marino, M. C., ... & Vawter, N. (2012). *10 PRINT CHR\$(205.5+RND(1)); : GOTO 10*. MIT Press. <https://doi.org/10.7551/mitpress/9040.001.0001>
- Navas, E. (2012). *Remix theory: The aesthetics of sampling*. Springer. <https://doi.org/10.1007/978-3-7091-1263-2>
- Newman, C. D., Alsuhaibani, R. S., Collard, M. L., & Maletic, J. I. (2017, February). Lexical categories for source code identifiers. In *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 228–239). IEEE. <https://doi.org/10.1109/SANER.2017.7884624>
- Nielebock, S., Krolikowski, D., Krüger, J., Leich, T., & Ortmeier, F. (2018). Commenting source code: Is it worth it for small programming tasks? *Empirical Software Engineering*, 24(3), 1418–1457. <https://dx.doi.org/10.1007/s10664-018-9664-z>
- Oliveira, D., Bruno, R., Madeiral, F., & Castor, F. (2020, September). Evaluating code readability and legibility: An examination of human-centric studies. In *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution* (pp. 348–359). IEEE. <https://doi.org/10.1109/ICSME46990.2020.00041>
- Panichella, S., Aponte, J., Di Penta, M., Marcus, A., & Canfora, G. (2012, June). Mining source code descriptions from developer communications. In *Proceedings of the 2012 20th IEEE International Conference on Program Comprehension (ICPC)* (pp. 63–72). IEEE. <https://doi.org/10.1109/ICPC.2012.6240510>
- Pascarella, L., Bruntink, M., & Bacchelli, A. (2019). Classifying code comments in Java software systems. *Empirical Software Engineering*, 24(3), 1499–1537. <https://doi.org/10.1007/s10664-019-09694-w>
- Peck, L., & Brown, S. W. (2024, May). Tool for constructing a large-scale corpus of code comments and other source code annotations. In *Proceedings of the 2nd Workshop on Computation and Written Language (CAWL) @ LREC-COLING 2024* (pp. 18–22). ELRA and ICCL.
- Peirce, C. S. (1985). Logic as semiotic: The theory of signs. In R. Innis (Ed.), *Semiotics: An introductory anthology* (pp. 4–23). Indiana University Press.
- Petersen, K., Feldt, R., Mujtaba, S., & Mattsson, M. (2008, June 26–27). Systematic mapping studies in software engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE)* (pp. 1–10). BCS Learning & Development. <https://doi.org/10.14236/ewic/EASE2008.8>
- Rahman, M. M., Roy, C. K., & Keivanloo, I. (2015, September 27–28). Recommending insightful comments for source code using crowdsourced knowledge. In *Proceedings of the 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (pp. 81–90). IEEE. <https://doi.org/10.1109/SCAM.2015.7335404>
- Sack, W. (2025). Interfacing programming language semantics and pragmatics: What does ‘Hello, World’ Mean? *Philosophies*, 10(4), Article 86. <https://doi.org/10.3390/philosophies10040086>
- Scott, M. L., & Aldrich, J. (2025). *Programming language pragmatics*. Elsevier.
- Searle, J. R. (1969). *Speech acts: An essay in the philosophy of language*. Cambridge University.
- Selic, B. (2008). Personal reflections on automation, programming culture, and model-based software engineering. *Automated Software Engineering*, 15(3), 379–391. <https://doi.org/10.1007/s10515-008-0035-7>
- Silverstein, M. (2003). Indexical order and the dialectics of sociolinguistic life. *Language & Communication*, 23(3–4), 193–229. [https://dx.doi.org/10.1016/S0271-5309\(03\)00013-2](https://dx.doi.org/10.1016/S0271-5309(03)00013-2)
- Spinuzzi, C. (2001, October). Software development as mediated activity: Applying three analytical frameworks for studying compound mediation. In *Proceedings of the 19th Annual International Conference on Computer Documentation* (pp. 58–67). ACM. <https://dx.doi.org/10.1145/501516.501528>
- Spinuzzi, C. (2003). Compound mediation in software development: Using genre ecologies to study textual artifacts. In C. Bazerman & D. R. Russell (Eds.), *Writing selves/writing societies: Research from activity perspectives* (pp. 97–124). WAC Clearinghouse. <https://dx.doi.org/10.37514/PER-B.2003.2317.2.03>

- Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., & Vijay-Shanker, K. (2010). Towards automatically generating summary comments for Java methods. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)* (pp. 43–52). ACM. <https://doi.org/10.1145/1858996.1859006>
- Sridhara, G., Hill, E., Pollock, L., & Vijay-Shanker, K. (2008, June). Identifying word relations in software: A comparative study of semantic similarity tools. In *Proceedings of the 2008 16th IEEE International Conference on Program Comprehension* (pp. 123–132). IEEE. <https://doi.org/10.1109/ICPC.2008.18>
- Steidl, D., Hummel, B., & Juergens, E. (2013, May 20–21). Quality analysis of source code comments. In *Proceedings of the 2013 21st International Conference on Program Comprehension (ICPC)* (pp. 83–92). IEEE. <https://doi.org/10.1109/ICPC.2013.6613836>
- Steinert, B., Taeumel, M., Lincke, J., Pape, T., & Hirschfeld, R. (2010, January 25–28). Codetalk conversations about code. In *Proceedings of the 2010 8th International Conference on Creating, Connecting and Collaborating through Computing* (pp. 11–18). IEEE. <https://doi.org/10.1109/C5.2010.11>
- Storey, M.-A., Zagalsky, A., Filho, F. F., Singer, L., & German, D. M. (2017). How social and communication channels shape and challenge a participatory culture in software development. *IEEE Transactions on Software Engineering*, 43(2), 185–204. <https://dx.doi.org/10.1109/TSE.2016.2584053>
- Swales, J. (1990). The concept of discourse community. In J. Swales (Ed.), *Genre analysis: English in academic and research settings* (pp. 21–32). Cambridge University Press.
- Tan, L., Yuan, D., Krishna, G., & Zhou, Y. (2007, October 14–17). /\* icodecomment: Bugs or bad comments? \*/ In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (pp. 145–158). ACM. <https://dx.doi.org/10.1145/1294261.1294276>
- Tian, Y., Zhang, Y., Stol, K. J., Jiang, L., & Liu, H. (2022, May). What makes a good commit message? In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)* (pp. 2389–2401). ACM.
- Treude, C., Barzilay, O., & Storey, M. A. (2011, May). How do programmers ask and answer questions on the web? (NIER track). In *Proceedings of the 33rd International Conference on Software Engineering* (pp. 804–807). ACM. <https://doi.org/10.1145/1985793.1985907>
- Tsay, J., Dabbish, L., & Herbsleb, J. (2014, May 31–June 7). Influence of social and technical factors for evaluating contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 356–366). ACM. <https://doi.org/10.1145/2568225.2568315>
- Van Valin, R. D. (2014). Role and reference grammar. In A. Carnie, D. Siddiqi, & Y. Sato (Eds.), *The Routledge handbook of syntax* (pp. 579–603). Routledge.
- Vinz, B. L., & Etzkorn, L. H. (2008, July 14–17). Comments as a sublanguage: A study of comment grammar and purpose. In *Proceedings of the 2008 International Conference on Software Engineering Research & Practice* (pp. 17–23). CSREA Press.
- Williams, A. (2003, October 12–15). Examining the use case as genre in software development and documentation. In *Proceedings of the 21st Annual International Conference on Documentation SIGDOC'03* (pp. 12–19). ACM. <https://doi.org/10.1145/944868.944872>
- Wolf, C. T. (2019, March). Professional identity and information use: On becoming a machine learning developer. In *Proceedings of the International Conference on Information* (pp. 625–636). Springer. [https://dx.doi.org/10.1007/978-3-030-15742-5\\_59](https://dx.doi.org/10.1007/978-3-030-15742-5_59)
- Yang, B., Liping, Z., & Fengrong, Z. (2019, January 12–14). A survey on research of code comment. In *Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences* (pp. 45–51). ACM.

## ABOUT THE AUTHOR

Elizaveta G. Grishechko

PhD in Linguistics, Senior Lecturer

Department of Foreign Languages, Faculty of Economics

RUDN University, Moscow, Russia

Postal address: 6 Miklukho-Maklaya Str., 117198 Moscow, Russia

Email: [grishechko-eg@rudn.ru](mailto:grishechko-eg@rudn.ru)ORCID ID: <https://orcid.org/0000-0002-0799-1471>