



The Semantic Workspace:
Augmenting Exploratory Programming
with Integrated Generative AI Tools

*Der semantische Workspace:
Augmentierung explorativer Programmierung
durch die Integration generativer KI-Werkzeuge*

by

Christoph Thiede

A thesis submitted to the
Hasso Plattner Institute
at the University of Potsdam, Germany
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE IN IT-SYSTEMS ENGINEERING

Supervisors

Prof. Dr. Robert Hirschfeld
Dr. Marcel Taeumel
Lukas Böhme, M.Sc.

Software Architecture Group
Hasso Plattner Institute
University of Potsdam, Germany

September 30, 2024

We provide the \LaTeX sources of this thesis at <https://github.com/LinqLover/semexp-thesis>. There, you can also find a two-sided version of this document for printing.

If you have any questions or feedback, or require a fully accessible version of this document, please create an issue in the GitHub repository or contact me at christoph.thiede@student.hpi.de.

*

Parts of this work will be published and presented at the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '24) [70].

Abstract

In the exploratory programming practice, programmers iteratively ask questions and run experiments to understand and develop software systems. However, traditional exploratory programming workflows often lead to distractions and information overload, as programmers need to handle numerous implementation artifacts. Meanwhile, semantic technologies—text generation using large language models (LLMs) and semantic retrieval using embeddings—are establishing themselves in other development practices to assist in writing and searching code.

We propose an augmented exploratory programming workflow that integrates semantic technologies into programming systems, allowing programmers to interact with them through more conceptual interfaces. Our semantic workspace introduces three semantic programming tools for augmenting and automating exploration: semantic suggestions anticipate the intentions of programmers and recommend possible experiments, semantic completions continue their plans through contextualized suggestions, and semantic conversations enable high-level, natural-language questions about objects.

Our semantic exploration kernel uses semantic technologies to power these tools with a suggestion engine for recommending and contextualizing artifacts, and with an exploratory programming agent for autonomous experiments and conversations with programmers. We explore embedding- and term-based strategies for searching and ranking artifacts, and we design prompts and system interfaces for an agent based on the GPT-4o model.

We successfully used a prototype of the semantic workspace for Squeak/Smalltalk to augment different exploratory activities. From our experience, semantic tools show promise in streamlining the exploratory programming workflow, but they must be further optimized to master exploratory practices and semantic understanding and to reduce time and resource consumption. We believe that our work is an important step toward a new era of semantic exploratory programming, where programmers and deeply intelligent agents collaborate effectively to comprehend and extend large systems.

Zusammenfassung (German Abstract)

In der Praktik des explorativen Programmierens stellen Programmierer*innen iterativ Fragen und führen Experimente durch, um Softwaresysteme zu verstehen und zu entwickeln. Traditionelle Workflows im explorativen Programmieren führen jedoch oft zu Ablenkungen und Informationsüberflutung, da Programmierer zahlreiche Implementierungsartefakte berücksichtigen müssen. Gleichzeitig etablieren sich semantische Technologien – Textgenerierung durch große Sprachmodelle (LLMs) und semantische Suche mit Embeddings – in anderen Bereichen der Softwareentwicklung, um das Schreiben und Suchen von Code zu unterstützen.

Wir schlagen einen erweiterten Workflow für das explorative Programmieren vor, der semantische Technologien in Programmiersysteme integriert und es Programmierenden ermöglicht, über konzeptuelle Schnittstellen mit diesen zu interagieren. Unser semantischer Workspace stellt drei semantische Werkzeuge zur Verfügung, um Explorationen zu augmentieren und zu automatisieren: Semantische Vorschläge antizipieren die Intentionen von Programmierenden und empfehlen mögliche Experimente, semantische Vervollständigungen setzen ihre Pläne durch kontextualisierte Vorschläge fort, und semantische Konversationen erlauben abstrakte, natürlichsprachliche Fragen über Objekte.

Unser semantischer Explorationskernel nutzt semantische Technologien, um diese Werkzeuge mit einem Vorschlagsmodul für Artefakte sowie einem explorativen Programmieragenten für autonome Experimente und Konversationen mit Programmierern anzutreiben. Wir untersuchen embedding- und termbasierte Strategien für das Suchen und Ranking von Artefakten und entwickeln Prompts und Systemschnittstellen für einen Agenten auf Basis des GPT-4o-Modells.

Wir haben einen Prototypen des semantischen Workspaces für Squeak/Smalltalk erfolgreich genutzt, um verschiedene explorative Aktivitäten zu augmentieren. Unsere Erfahrung zeigt, dass semantische Werkzeuge vielversprechend sind, um den Workflow des explorativen Programmierens zu vereinfachen. Sie müssen jedoch weiter optimiert werden, um explorative Praktiken und semantisches Verständnis besser zu beherrschen und den Zeit- und Ressourcenverbrauch zu reduzieren. Wir glauben, dass unsere Arbeit einen wichtigen Schritt in Richtung einer neuen Ära des semantischen explorativen Programmierens darstellt, in der Programmierende und tiefgreifend intelligente Agenten effektiv zusammenarbeiten, um große Systeme zu verstehen und zu erweitern.

Acknowledgments

While I started writing this thesis six months ago, much of the work presented here originates from the past one and a half years, or even before that, considering my broader orientation within the exploratory programming field. I am grateful to several people who made this research not only an insightful but also a very fulfilling experience.

First and foremost, to Marcel Taeumel for mentoring and advising me throughout this research as well as several other projects before. Thank you for collaborating on multiple papers; for sharing your experience and ideas about exploratory programming, academia, and Squeak during many extensive and delightful discussions; for providing me with *significant* feedback on my writing, always lending me an ear or eye when I bombarded you with too many questions, and critically shaping my entire master's studies into something wonderful. And for allowing me to adopt your beautiful drawing style in this thesis! To Lukas Böhme and Toni Mattis for contributing their ideas and perspectives on machine learning to my research, and again to Lukas for reviewing my writing. To Robert Hirschfeld for motivating me to study this field, making this research possible (including financial support), and helping me on various other matters.

I am also thankful to other members and students of the SWA group for discussing my ideas; to the three anonymous reviewers of the Onward! paper for providing their inspiring and detailed reviews; and to u/plasticpears for discussing their vision of AI-augmented exploratory programming systems with me.¹ I also thank Patrick Rein and Willy Scheibel for the opportunity to work together on earlier publications: Patrick for showing me the first steps of educational writing and Willy for sharing an elaborate mindset of academic thinking and writing with me.

This thesis would not look the same without the ecosystem and the individual support by Tobias Pape and the T_EX.SE community.

Finally, I would like to thank my family for reading my thesis and for all your other invaluable support.

Carpe Squeak!

¹https://www.reddit.com/r/smalltalk/comments/1b3dx4q/smalltalk_llms

Contents

1. Introduction	1
2. Background	5
2.1. Exploratory Programming	5
2.2. Exploratory Programming Systems	8
2.3. Challenges in Exploratory Programming Systems	13
2.4. A New Opportunity: Semantic Technologies	15
3. Approach	23
3.1. The Augmented Exploratory Programming Workflow	23
3.2. The Semantic Workspace	25
3.2.1. Semantic Suggestions	27
3.2.2. Semantic Completions	30
3.2.3. Semantic Conversations	32
4. The Semantic Exploration Kernel	34
4.1. Architecture	34
4.2. A Blackboard Framework for Managing Suggestions	37
4.3. Semantic Object Interfaces for Exploratory Programming	39
5. Suggesting Experiments with Semantic Retrieval	46
5.1. Artifacts and Strategies	46
5.2. Semantic Search	48
5.3. Semantic Ranking	51
5.4. Semantic Generation	55
6. Building an Exploratory Programming Agent with GPT-4o	57
6.1. Implementing Policies through Prompts	57
6.2. Designing System Interfaces for Automatic Agent Experiments	58
6.3. Optimizing for Semantic Completions	61
7. Integrating the Semantic Workspace into Squeak	64
7.1. Semantic Suggestions	64
7.2. Semantic Completions	66
7.3. Semantic Conversations	67

Contents

8. Applications	69
8.1. Prototyping User Interfaces with Semantic Suggestions and Completions	69
8.2. Exploring Text Formatting through Semantic Conversations . .	73
8.3. Toward a Semantic Toolset for Exploratory Programming	75
9. Discussion	81
9.1. Potential of Semantic Technologies	81
9.2. Performance of Semantic Tools	85
9.3. Exploratory Programming Experience	89
9.4. Ethical Considerations of Semantic Technologies	94
10. Related Work	95
10.1. Suggestions in Programming Systems	95
10.2. High-Level Programming Interfaces	96
11. Conclusion	98
A. Accessing Semantic Technologies through the SEMANTICTEXT Framework	101
A.1. Domain Model	101
A.2. Language Providers	105
A.3. Tool Support	106
B. Example Conversation Log of the Exploratory Programming Agent	110
C. Recommendations for Tool Developers	121
Bibliography	123

List of Figures and Tables

2.1.	A single <i>research process</i> in the <i>exploratory programming workflow</i> .	6
2.2.	A <i>hierarchical research process</i> in the <i>exploratory programming workflow</i> .	7
2.3.	A collection of <i>exploratory programming tools</i> in the Squeak/Smalltalk system.	11
2.4.	A high-level overview of the <i>transformer architecture</i> for large language models that embed and generate text.	17
3.1.	Our model of an <i>augmented exploratory programming workflow</i> .	23
3.2.	The support spectrum of different <i>semantic tools</i> in the semantic workspace.	26
3.3.	Our concept of a <i>semantic suggestions</i> tool in the semantic workspace.	28
3.4.	Our concept of a <i>semantic completions</i> tool in the semantic workspace.	30
3.5.	Our concept of a <i>semantic conversations</i> tool in the semantic workspace.	32
4.1.	The high-level architecture of the <i>semantic exploration kernel</i> and its environment.	35
4.2.	The general object model of our <i>blackboard framework</i> for creating suggestions.	37
4.3.	The involved artifacts and scheduled strategies for invoking the blackboard framework to find correlated methods for a given input method and notes.	39
4.4.	Our approach of <i>semantic object interfaces</i> for semantic exploratory programming systems.	40
4.5.	Possible integration of a conversational semantic interface into a traditional object inspection tool.	44
5.1.	Merge graph of all available artifact types and strategies in the suggestion engine.	47
5.2.	Comparing semantic ranking methods against different objectives.	54
6.1.	Schematic prompt design for conversations of the exploratory programming agent with the user and the system.	59

List of Figures and Tables

6.2.	The function interface that connects our exploratory programming agent to the Squeak/Smalltalk system.	60
6.3.	Optimized prompt design of the exploratory programming agent for generating stage-1 code completions.	62
6.4.	Optimized prompt design of the exploratory programming agent for generating stage-2 code completions.	63
7.1.	The user interface of our <i>suggestion space</i> for displaying semantic suggestions in Squeak.	65
7.2.	Integration of semantic completions into the <code>AUTOCOMPLETION</code> tool for Squeak.	66
7.3.	Integration of semantic conversations into Squeak’s inspector.	67
8.1.	Bootstrapping the prototyping of a Morphic counter application in a workspace.	69
8.2.	Using the suggestion space to browse relevant protocols of the class <code>Morph</code> for prototyping a user interface.	70
8.3.	Using the semantic autocompletion to explore different customization possibilities for borders of our UI prototype.	71
8.4.	The complete prototype of our counter application in Squeak.	72
8.5.	Using the semantic conversation mode in an inspector to chat with a formatted <code>Text</code> object.	74
8.6.	Sending a semantic message to search and filter a small project management system for common-sense questions.	75
8.7.	Sketching a generic integration of semantic object interfaces into object-oriented user interfaces.	76
8.8.	In a semantic debugger, programmers can ask for the origin and meaning of values on the program stack.	77
8.9.	In a semantic code browser, programmers can engage in natural-language conversations with classes to explore them.	78
8.10.	In a semantic projectional editor, programmers can chat with single code blocks to explain, refactor, or execute them.	79
9.1.	Main response times and memory consumptions for the tools in our prototype of the semantic workspace.	85
A.1.	The object model for semantic retrieval in <code>SEMANTICTEXT</code>	102
A.2.	The object model for conversational agents in <code>SEMANTICTEXT</code>	104
A.3.	The <i>conversation editor</i> for accessing and manipulating conversations with LLM agents.	108
A.4.	Through <i>expense watchers</i> , programmers can track the costs of semantic expenses within the programming system.	109

1. Introduction

The important thing is not to stop questioning. Curiosity has its own reason for existing.

— Albert Einstein

In the realm of software development, curiosity is essential for understanding problems and finding solutions. Programmers spend a large amount of time searching for useful interfaces, concepts, and methods—and they spend a similar amount of time understanding systems, plugging concepts together, and experimenting with prototypes. All of this involves raising many different questions: *which parts of this system do I need to touch to add a new feature? What information does this object contain, and how can I access or manipulate it? Why has this cache not been reset—how have others solved this problem? What would it look and feel like if we placed this button there?*

All this need for knowledge can be expressed as *questions*, and we can say programmers have *conversations* with the system in which they conduct *experiments* to collect the information required to answer these questions [61]. Good conversations are vivid, rapid, rich: many answers lead to new thoughts, revised questions, alternative approaches—and with every further question, programmers gradually develop their understanding of the systems and problems they are working with, until they achieve a viable solution and their curiosity is satisfied for the moment.

Exploratory programming systems [45, 54] are designed to support such conversations by offering several *tools*, through which programmers can interact with software systems (or their parts) to answer questions. One prominent exploratory programming system is *Squeak/Smalltalk* [17, 67], where *objects* serve as the main building block for defining systems. Objects execute behavior by sending messages to each other, and they handle messages by accessing and changing their internal state. Through tools of the Smalltalk system, programmers can explore and develop the objects in a running system by inspecting and modifying their state and behavior.

In Squeak, a central tool for exploratory programming is the *workspace*: here, programmers can write and evaluate scripts to prototype new solutions, browse the classes of objects to research and discover existing implementations and protocols, or inspect particular objects and send them messages to explore their state and capabilities.

The Cost of Questioning

However, exploratory conversations can be expensive. Even simple questions might demand the full attention and structured thinking from programmers: *when has this order been filed? I can find that out by inspecting this object. Ah, it has a `creationDate` field, but oh no, why is this just a plain number? It could be a Unix timestamp. How can I convert that into a human-readable representation? Is there any method on `Date` that does this for me? Seems not so. Maybe `DateAndTime`? Yes, `DateAndTime` `fromUnixTime`: `creationDate` should do the job ... but wait, surely this order has not been created in year 56170? Is this actually a millisecond timestamp? Okay, so I can divide it by 1000 and try again ... fix that syntax slip ... alright, so this order has been filed on March 14th this year. What was I going to do again?*

We note two major challenges in the exploratory programming workflow that traditional exploratory programming systems have not yet found sufficient solutions for: *semantic distances* and *information overload*. First, programmers regularly face substantial *semantic distances* between their high-level intentions and the low-level interfaces of systems. Programmers have to manually bridge these distances by translating conceptual questions into technical operations and interpreting technical results in terms of their mental model.

Second, they face *information overload* as the number of possible information sources (such as comprehensive interfaces, documentation, and examples), as well as the number of possible experiments and prototypes, often exceed human capabilities for information processing within limited time. Thus, programmers are forced to consider only a small, often less relevant fraction of the available data. If they nevertheless decide to skim all available information, they are frequently *distracted* by irrelevant artifacts while trying to filter them.

All these mental overheads and distractions impede exploratory conversations and restrain programmers from maintaining their flow [13], exhausting expedient options, and finding feasible solutions.

AI to the Rescue?

At the same time, the recent progress of artificial intelligence (AI) and machine learning (ML) methods has already supported users and programmers at bridging barriers and handling extensive information in various domains. Search and recommendation algorithms on online marketplaces, social-media platforms, and streaming services use *document embeddings* [81, 97] to find objects based on their *semantics* or meaning. Generative AI tools such as ChatGPT¹, Claude² and others [101, 107, 112] employ *large language models* (LLMs) to help people create, automate, and learn.

1. Introduction

In particular, many programmers have started to leverage generative AI for software development: millions of developers generate code with the help of code completion tools such as GitHub Copilot³ Tabnine⁴ and others, or they use conversational agents such as GitHub Copilot Chat to explore and modify code bases. Similarly, tools such as Microsoft IntelliCode⁵ and Sourcegraph⁶ employ embeddings to find and rank relevant source code snippets.

We subsume both technologies—*semantic retrieval* using document embeddings and *text generation* using LLMs—under the term *semantic technologies*.

Research Objective and Structure

We believe that semantic technologies have significant potential to support exploratory programmers in processing extensive information and bridging semantic distances. This leads us to our research question:

How can we augment the exploratory programming workflow by integrating semantic technologies into exploratory programming systems?

That is, we want to study possible applications of semantic technologies for exploratory programming, investigate tool designs and interaction mechanisms to give programmers simple access to these technologies, and collect practical considerations for the design and implementation of such tools.

In answer to this question, our thesis statement reads as follows:

By tightly integrating *semantic interfaces* into exploratory programming systems through which programmers can easily delegate flexible portions of their workflow to *intelligent agents* and thus closely *collaborate* with semantic technologies, we can augment and streamline the exploratory programming workflow.

For this, we propose a model for an *augmented exploratory programming workflow* in which the exploratory activities of programmers are tracked and anticipated, and in which programmers are enabled to access and interact with AI-suggested experiments and results through a *semantic workspace*. We believe that our approach will allow tool developers to design tools for exploratory programming systems that improve the experience of programmers by enabling them to collaborate with semantic technologies.

¹<https://chat.openai.com/>

²<https://claude.ai/>

³<https://github.com/features/copilot>

⁴<https://www.tabnine.com/>

⁵<https://visualstudio.microsoft.com/services/intellicode/>

⁶<https://sourcegraph.com/>

1. Introduction

To study this hypothesis, we make the following contributions:

1. We propose the *augmented exploratory programming workflow*, which captures the traditional process and challenges of exploratory programmers and describes possible starting points for integrating semantic technologies.
2. We present a new *semantic workspace* as a conceptual design for exploratory programming systems that support an augmented exploratory programming workflow by providing new or enhanced *semantic interfaces* for programmers.
3. We describe our architecture of a *semantic exploration kernel* with different components that use semantic technologies to augment exploratory programming tools, and we implement a prototype for Squeak/Smalltalk using language models by OpenAI.

We make all artifacts of our research available in a public GitHub repository:⁸

We organize the work presented in this thesis as follows:

CHAPTER 2 introduces theoretical foundations behind exploratory programming systems and semantic technologies and describes how the challenges of the former may be addressed by the opportunities of the latter.

CHAPTER 3 presents our model of the augmented exploratory programming workflow and describes our conception of a semantic workspace.

CHAPTER 4 describes the high-level architecture of the semantic exploration kernel and explains its fundamental components: a *suggestion engine* and a *semantic exploratory agent*.

CHAPTER 5 discusses the application of semantic retrieval methods for suggesting experiments.

CHAPTER 6 describes our construction of an exploratory programming agent for implementing conversational interfaces and automating experiments.

CHAPTER 7 sketches the implementation of our prototype by integrating it with the Squeak/Smalltalk programming system and our SEMANTICTEXT framework for semantic technologies that uses OpenAI’s language models.

CHAPTER 8 illustrates applications of the semantic workspace through different case studies.

CHAPTER 9 discusses opportunities and challenges of semantic technologies for augmenting the exploratory programming workflow regarding technical feasibility, programming experience, and ethical concerns.

CHAPTER 10 provides an overview of related work in the fields of exploratory programming and semantic technologies.

CHAPTER 11 summarizes our results and discusses possible directions for future work.

⁸<https://github.com/hpi-swa-lab/SemanticSqueak>

2. Background

In this chapter, we introduce the theoretical foundations of the *exploratory programming* practice and provide a model of the *exploratory programming workflow*. We describe how *exploratory programming systems* such as Squeak/Smalltalk support this practice through different interaction mechanisms and tools, and identify existing challenges related to the *semantic immediacy* and *information overload* that current exploratory programming systems impose on the workflow of exploratory programmers.

We explain two *semantic technologies*—*semantic retrieval* and *generative large language models*—that leverage AI and ML methods to process and generate text based on their abstract meaning. Finally, we briefly sketch how these technologies could be used to meet the challenges of current exploratory programming systems and augment the exploratory programming workflow.

2.1. Exploratory Programming

Exploratory programming is a software engineering practice that promotes a notion of projects where programmers have a rather emergent than upfront understanding of the problem domain [23, 45, 54]: for example, they might not know how a new user interface should look and feel to achieve a good user experience, or they might be unaware of the facilities and limitations that an existing code base or framework provides. To acquire such knowledge, the exploratory programming practice tightly intertwines reverse engineering, prototyping, and testing in an iterative manner [61].

In our model of an *exploratory programming workflow*, we employ a metaphor from ordinary science: exploratory programmers are like scientists in the project domain who apply the scientific method and iteratively refine their comprehension of both the problem domain and the solution domain. To refine their comprehension, they ask questions and find answers to them. Figure 2.1 displays a single instance of a *research process* in our exploratory programming workflow model.

Finding answers requires programmers to descend from the conceptual level of the original questions (their *mental model*) into a lower abstraction level where they can dissect questions into their underlying terms, concepts, and technical foundations (the *technical model*). Here, they interact with the underlying systems by planning, executing, and evaluating a—potentially

2. Background

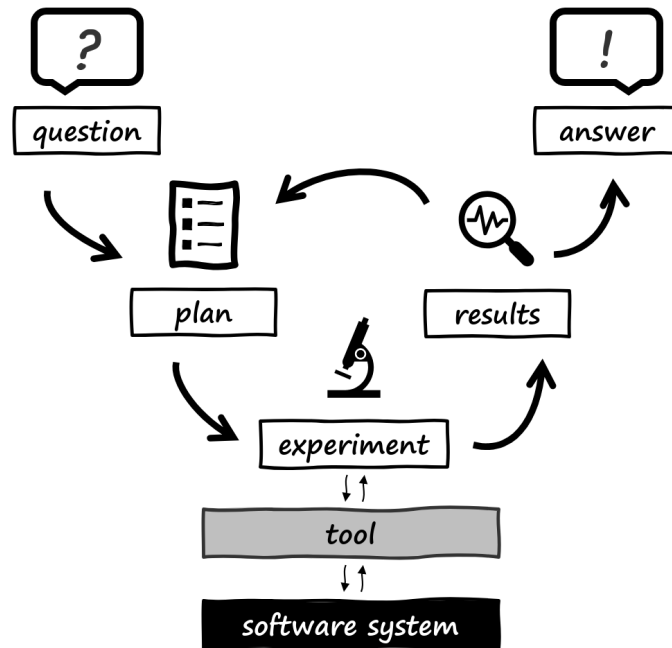


Figure 2.1.: A single research process instance in the exploratory programming workflow. Exploratory programmers start with a question about the system. To answer the question, they plan one or multiple experiments, execute them by interacting with the system through tools, and observe their results. They repeat until they have acquired enough information and then combine it to answer the initial question.

large—number of *experiments* and repeat until they are able to answer the original question. Experiments include a wide range of activities aimed at generating knowledge: for example, programmers can research information in the documentation or implementation of a system or related communication platforms; inspect objects of a system at runtime to understand their internal state; run scripts to test interfaces and observe their effect; build and test prototypes; and so on.

For nontrivial questions, the research process often spans multiple abstraction levels, as the experiments that programmers plan to answer high-level questions are still too abstract to directly communicate them to the system. Thus, programmers need to handle abstract experiments as new, subordinate questions and answer them on a lower abstraction level. This leads to a *hierarchical research process* in which programmers gradually descend into the implementation details of the subjects of questions until they reach the interfaces of the system through which they can execute technical experiments. [Figure 2.2](#) displays a complex research process that exploratory programmers walk through when answering a high-level question.

2. Background

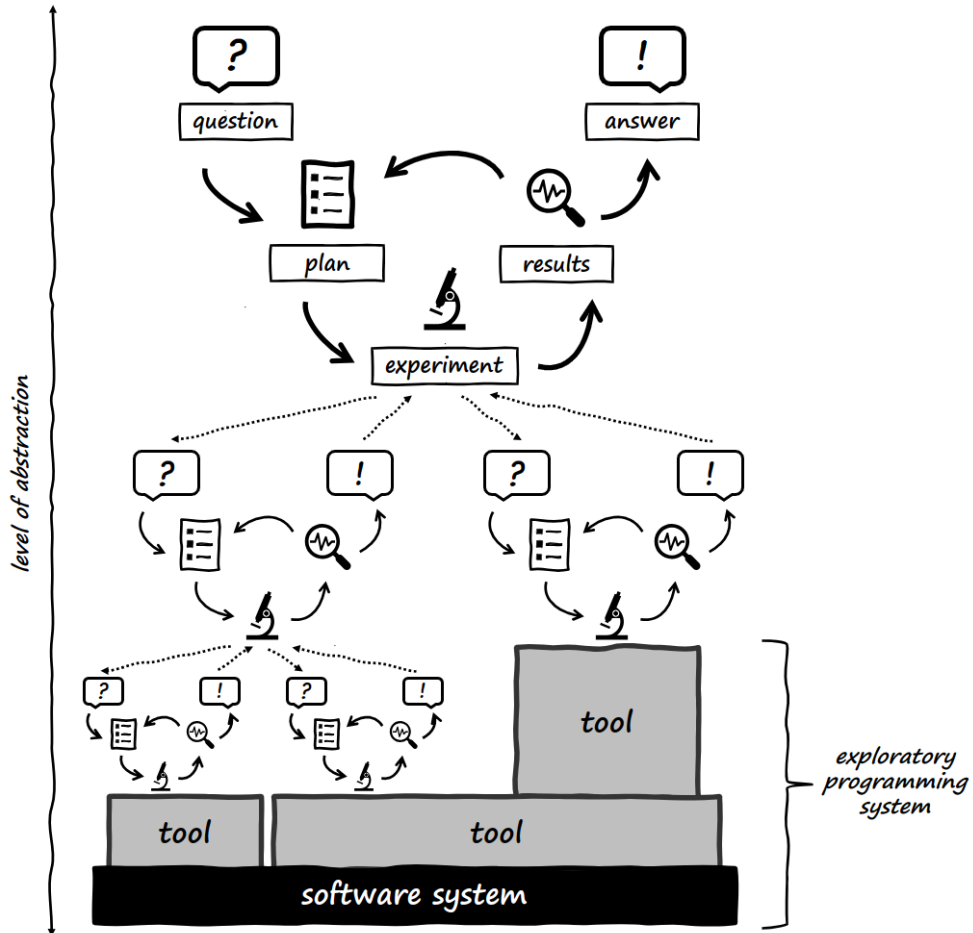


Figure 2.2.: A complex, hierarchical research process in the exploratory programming workflow. To answer a high-level question about the system, programmers need to bridge multiple levels of abstractions by incrementally breaking down experiments into more concrete questions and answering those until they reach the interfaces of the exploratory programming system. The programming system consists of different low-to-medium-level tools for interacting with the software system.

Example. A programmer might wish to design a new user interface (“How could a convenient UI look?”). To answer this question, they plan to experiment with different UI concepts that they want to test through small prototypes (“Let’s build a tree-based or pane-based interface and play around with it!”).

However, to execute these experiments, the programmer needs to perform subordinate research processes in which they conduct further experiments to implement and test individual prototypes (“How can I create a tree widget? How can I retrieve the required data? How many interactions are required to navigate through this UI?”).

Recursively, some of these research processes might yield further experiments that need to be broken down before they can be communicated to the system (“What classes does this package provide to build widgets? What messages does this class understand? What usage patterns do existing users of this class show?”).

2.2. Exploratory Programming Systems

As a consequence of our exploratory programming workflow, exploratory programmers crucially depend on interfaces to the system under exploration to execute experiments. For example, these interfaces allow programmers to inspect variables, send messages to objects, or browse and write code. We refer to the set of all interfaces and tools that provide such access to software systems as an *exploratory programming system*, which serves as the foundation and facilitation of all exploratory programming activities [15, 23, 54, 61]. These interfaces and tools can support the exploratory programming workflow at different levels of abstraction: programmers can perform lower-level operations by executing scripts, view and modify objects through domain-specific representations [10], or use task-specific tools for searching and filtering source code [62].

One important property of exploratory programming systems is *liveness*, which allows programmers to explore and modify systems interactively, that is, with short feedback cycles [32, 45, 64]. To support such live programming, exploratory programming systems commonly include mechanisms to develop systems at runtime without restarting or recompiling them after every change.

Examples of exploratory programming systems include operating system shells such as the Linux Bash Shell [43], through which programmers can dynamically navigate and configure Linux systems, computational notebooks such as Jupyter [57], which allow programmers to explore data and prototype algorithms incrementally, and Smalltalk environments [17], in which

2. Background

programmers can develop and debug object-oriented systems at small and large scales.

Smalltalk and Squeak

Smalltalk is a programming system that especially promotes exploratory programming through its interactive programming environment and rich ecosystem of tools [22].

As a language, Smalltalk pursues simplicity in syntax, code organization, and its execution model [17]. Smalltalk is a strictly object-oriented language: everything is an object, and everything happens through message sending between objects. Every object is defined by its identity that distinguishes it from all other objects in the system, its internal (encapsulated) state, and its observable behavior that is implemented by methods that process messages. Smalltalk is *class-based*, meaning that (in contrast to prototype-based languages such as JavaScript), every object is an instance of a class (which is another regular object), and object behavior is usually defined as methods on classes and organized within *protocols* (also referred to as *message categories*).

As an environment, Smalltalk distinguishes itself by its interactive, image-based, and self-sustained architecture [16]. Smalltalk systems are *interactive*, allowing programmers to inspect and modify any object in the system at runtime. They are *image-based*, meaning that the whole system state (i.e., the object graph) is serializable and can be saved to and restored from a single file, called an *image*. They are *self-sustained*, as a vast majority of system concepts are implemented in the system itself and thus are explorable and malleable by programmers. For example, programmers can inspect and modify the compiler, debugging tools, or the user interface of the system at runtime [60].

Squeak is a modern, portable implementation of Smalltalk that extends the original Smalltalk-80 system with a rich toolset and the *Morphic* framework for tangible direct-manipulation user interfaces [21, 22, 67]. Through Morphic, programmers can build and modify graphical user interfaces interactively, explore and debug the implementation of applications, and manage multiple tasks and projects in parallel.

The Squeak Toolset



Exploratory programming systems such as Squeak provide different kinds of interfaces and tools, through which programmers can execute experiments in traditional exploratory programming systems to answer questions. While we focus on the Squeak/Smalltalk ecosystem in our overview of such tools, many tooling concepts have also been adopted by other exploratory programming systems, and some have been influenced by other systems.

2. Background

In Squeak, one central tool is the *workspace*, which lays the foundation for various exploratory programming questions [16, chap. 6; 67, sec. 1.4]. In the workspace, programmers interact with the system through a text-based interface by typing and evaluating code snippets (so-called *do-its*) to send messages to objects or retrieve variables. Similarly to read-eval-print-loop (REPL) interfaces or computational notebooks, workspaces can preserve prior results and objects for reuse through later do-its, fostering a conversational interaction style between the programmer and the system [61]. Beyond that, programmers can use workspaces to take and organize arbitrary notes such as to-do items in their programming system.

Due to their technical nature, most do-its can be seen as experiments at a low level of abstraction, as programmers have to write syntactically valid code and know the technical protocols of the objects they are interacting with. However, workspaces can also serve as a starting point for better integrated exploratory programming sessions through an extensible set of connected tools, which support programmers at higher levels of abstraction in their research process through domain-specific and task-specific interfaces (fig. 2.3).

Object inspection tools Through inspection tools, programmers can explore and modify the internal state of any object in the system. The standard inspection tools in Squeak are the *inspector* and *explorer*, which provide low-level access to the list of instance variables or fields in an object [16, chap. 8; 67, sec. 6.3].

However, inspection tools can also provide higher-level, domain-specific interfaces to objects: for example, *collection inspectors* display and manipulate the effective elements of different types of collection objects independently of their internal data structure; other systems such as GLAMOROUS TOOLKIT's *moldable inspector* promote domain-specific visualizations of objects such as charts and graphs [10]. Thus, programmers can directly find answers to more conceptual questions such as "How are these nodes connected?" without writing technical do-it scripts on their own.

Code browsing tools Through browsing tools, programmers can explore and modify the implementation, protocols, and documentation of classes [67, sec. 6.2]. In Squeak, different code browsers allow programmers to navigate and search software systems along their organization and inner relationships: *system browsers* provide access to classes through package structures or class hierarchies [16, chap. 9]; *message traces* and similar tools can be used to explore methods through different *usage graphs* [16, chap. 10; 69]. For example, the *senders/implementors* mechanism allows programmers to browse an (approximate¹) static call graph of methods that send or implement a given message name (also referred to as *selector*); other types of graph queries include ref-

2. Background

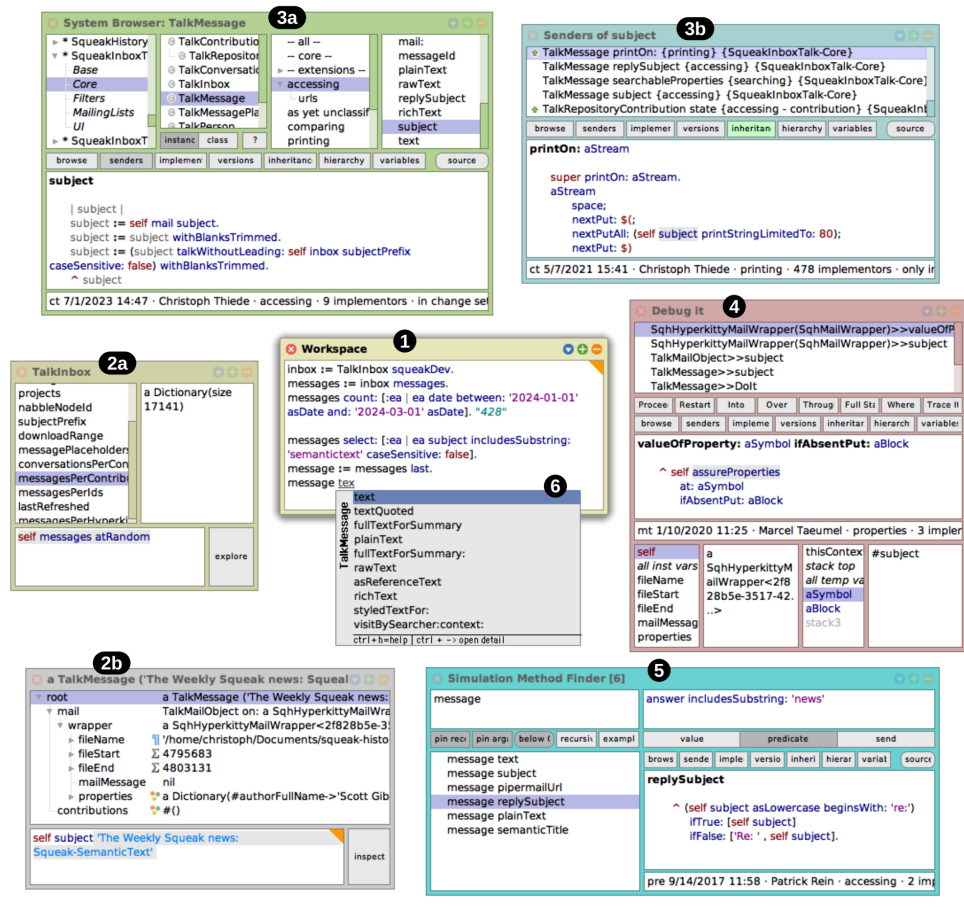


Figure 2.3.: A collection of different interconnected tools in the exploratory programming system Squeak/Smalltalk. At the heart is the workspace ①, through which programmers can execute *do-it* code snippets to retrieve information from objects. From the workspace, programmers can invoke other tools such as *object inspection tools* to examine the variables (②a) inspector) and nested properties (②b) explorer) of particular objects; *code browsing tools* to study the implementation of methods (③a) system browser) or their users (③b) message trace); *debugging tools* to explore method behavior in context (④ debugger); *symbolic execution tools* to find messages through constraints (⑤ simulation method finder^a); or *code completion tools* to browse available protocols in an editor context (⑥ Autocompletion^b). Each of these tools can assist programmers in different parts of their exploratory process by providing domain- or task-specific support.

^a<https://github.com/LinQlover/SimulationStudio>

^b<https://github.com/LeonMatthes/Autocompletion>

2. Background

erences and assignments to variables, methods that contain certain literals (constants), and a full-text search through source code and comments. In general, for any code objects (such as a class or method), we employ the term *users* to address its successors in a usage graph (such as its senders or dependent classes).

Thus, code browsing tools provide means for programmers to answer questions about the organization and implementation of systems through task-specific views and query interfaces.

Debugging tools In the first place, (symbolic) debuggers were designed to allow programmers to examine faulty programs by executing them step-by-step and finding the underlying cause of a bug [17, sec. 17.4; 16, sec. 18f.]. However, debuggers have also proven to be useful tools for exploring the behavior of software systems beyond bug-fixing, as they provide a concrete context for comprehending abstract source code and navigating through an actual call graph of a program [67, sec. 6.4]. Many tools also support *programming in the debugger* to modify the behavior of a running program, which allows for a “programming into existence” practice for interface-first, iterative prototyping [52]. *Babylonian programming* environments promote a similar practice by integrating the execution context of different examples into regular code browsing tools [42].

Back-in-time debuggers (also referred to as *time-travel debuggers* or *omniscient debuggers*) record the execution of a program or re-run it repeatedly to allow programmers to explore it independently of the original execution order [29, 39, 40]. On top of such an omniscient perspective on programs, they offer different interfaces for navigating programs along objects, dataflows [30], or state changes [71, 72] or for visualizing the program execution [12, 68]. For example, the WHYLINE approach provides a query interface where programmers can combine prepared natural-language blocks to ask a range of questions about the causes of certain events in a program [24].

Symbolic execution tools *Symbolic execution engines* make it possible to execute programs and send messages to objects despite the lack of concrete context by substituting unknown state with symbols and executing all possible code paths concurrently [9, 66]. This provides a basis for exploratory programming tools that can search the behavior of objects through speculative execution. Squeak’s *method finder* allows programmers to find unknown methods by specifying constraints over the inputs and outputs of message sends [67,

¹ Messages in Smalltalk are dispatched dynamically, and traditional Smalltalk systems do not have a typing system, preventing them from accurately predicting which methods a message send in a code snippet might activate.

2. Background

sec. 1.8].^{2,3} Tools such as INTELLITEST⁴ or SED [20] provide programmers with an overview of different examples and code paths for methods.

Thus, symbolic execution tools allow programmers to research the methods of objects by asking questions about their effective behavior without browsing or manually executing them.

Code completion tools Autocompletion tools support programmers at writing code by automatically suggesting possibly relevant message names and variables next to the text caret. Suggestions can be contextualized in various degrees, e.g., by considering the syntactic context of existing code, usage statistics from the current programming session [48], or information from runtime or typing systems [19]. Many modern code completion tools also employ large language models to suggest completions of entire statements or methods (see section 2.4).

Beside accelerating typing, code completion tools also support the research process of exploratory programmers by providing them with possible answers to questions about the usage of protocols (“What messages can I send to this variable?”) and making manual research through code browsing tools superfluous in some cases [4].

*

By using and combining different kinds of tools offered by exploratory programming systems, programmers can derive more informed answers to questions and delegate low-level parts of their research process to the system.

2.3. Challenges in Exploratory Programming Systems

Despite the existing tool support of exploratory programming systems, programmers frequently experience interruptions in their workflow as they have to switch between different abstraction levels of the research process when answering questions. While exploratory programming tools make it possible to delegate different questions to the system, this support is usually limited

²While Squeak does not include a symbolic execution engine at the time of writing, the method finder employs equivalent brute-force strategies to resolve the provided constraints.

³Another version of Squeak’s built-in method finder, which supports more generic types of symbolic queries, is available through the SIMULATIONSTUDIO package. Christoph Thiede: “Method Finder 2.” The general-purpose Squeak developers list, 2022-09-20. URL: <https://web.archive.org/web/20240630205751/https://lists.squeakfoundation.org/archives/list/squeak-dev@lists.squeakfoundation.org/thread/N40M3BYARXM0FDF40NM7IZWR7727WU2M/>.

⁴<https://web.archive.org/web/20240530170603/https://learn.microsoft.com/en-us/visualstudio/test/intellitest-manual/?view=vs-2022>

2. Background

to lower levels of abstraction or to specific domains, leaving programmers to address the majority of the research process on their own. As a consequence, programmers are distracted by many context switches, suffer from higher mental load, and might lose their flow [13].

To maintain the flow of programmers, exploratory programming systems should be designed to support their *experience of immediacy* by reducing their perceived *distances* in three dimensions [74]:

TEMPORAL IMMEDIACY: “Human beings recognize causality without conscious effort only when the time between causally related events is kept to a minimum.”

SPATIAL IMMEDIACY: “[...] means the physical distance between causally related events is kept to a minimum.”

SEMANTIC IMMEDIACY: “[...] means the conceptual distance between semantically related pieces of information is kept to a minimum.”

Spatial and temporal distances can often be managed by carefully and holistically designing (visual) user interfaces or engineering efficient algorithms. On the other hand, we argue that semantic distance presents a greater—and largely unmet—challenge to tool developers. This is because traditional tools are not capable of *understanding* the underlying concepts and semantics of the information they display and process and thus are limited in their ability to associate different pieces of information. Similarly, they lack knowledge and comprehension of the higher-level questions and goals of programmers, making it impossible in many situations to transfer required information into the desired context.

The necessary mapping between the technical model of systems and the mental model of programmers can be described as overall design challenges through the two *gulfs of execution and evaluation* [37]: the gulf of execution references the programmers’ challenges to express their questions as inputs to the system’s interface, while the gulf of evaluation represents their challenges to translate the outputs from the system back into their mental model. In exploratory programming, the gulf of execution typically includes activities such as planning and conducting experiments by using tools or writing code, while the gulf of evaluation contains tasks such as comprehending and summarizing technical outputs like comprehensive lists or long and complex source code [46].

Example. A programmer might wish to build a prototype to test a new UI concept (“How would this UI look and feel?”). Despite they already have a good idea of the visual appearance, they are burdened with expensive implementation work: they have to write syntactically valid code, browse the available packages and protocols of a UI framework to find required class and method names, and fix any bugs that they accidentally created in their implementation until they achieve a testable prototype.

Analogously, researching existing solutions that solve problems similar to the current matter of programmers involves a lot of manual labor and low-level interactions with systems. Traditional code search tools such as Squeak’s message trace primarily provide technical interfaces that only take a syntactic perspective on source code. For instance, programmers might want to find methods that implement a certain concept, but message traces only offer lexical search functions, which cannot identify concepts or find synonyms. Programmers might want to exclude certain kinds of solutions (such as tests or special metaprogramming code), but message traces can only filter methods based on their class and package names. Programmers might want to sort results by their conceptual relevance to their overarching question, but most code search tools are limited to popularity- or recency-based sort functions. Thus, programmers spend a lot of their research time manually identifying results as “obviously” irrelevant rather than investigating and learning from relevant samples.

Therefore, the answers that exploratory programmers can gain within a limited amount of temporal and mental resources are limited in both their quantity and quality, as the support by exploratory programming systems is tied to the technical level of information, reducing their programming experience. We identify a need for exploratory programming systems that improve semantic immediacy for programmers by working more closely with the mental model and overarching context of programmers and by providing more conceptual, “intelligent” tools that are able to handle or support high-level questions of programmers.

2.4. A New Opportunity: Semantic Technologies

At the same time, different AI methods for information processing have gained popularity over the last few years, which promise better user support in various domains—including programming—by approaching information based on its semantics. Concretely, we use the term *semantic technologies* to refer to two

2. Background

methods: *semantic retrieval* using document embeddings and *text generation* using LLMs.

Both technologies approach the semantic content of textual data through the concept of *embeddings*. An embedding is a numeric representation of a *document* (such as a single word, a (source) text, or also another embeddable object such as a picture [82]) through a high-dimensional vector whose components describe the relatedness of the document to different *features* [81, 97]. Features can represent arbitrary kinds of properties or topics, such as (programming) language, sentiment, or length, but in many settings, they are used to encode continuous combinations of various concepts through dimensional reduction. In this setting, the distance between two embeddings in the vector space indicates the similarity of their associated documents with regard to the employed concepts. This lays the foundation for performing different arithmetic operations to compare, organize, or sort documents in a given context.

Approaching a definition. What is semantics?

Semantics (from Ancient Greek *sēmantikós*, giving signs) refers to the significance or meaning of information in a document. It is often opposed to its counterpart *syntactics* (from Ancient Greek *śyntaxis*, assembling together), which describes the structural arrangement and grammatical rules of a document or fragment thereof.

To understand the semantics of a document, a rational agent (whether human or machine) typically has to employ different kinds of *context*: a comprehensive understanding of the used language(s), any surrounding information (such as an ongoing conversation or programming session), and relevant *concepts* (ranging from common sense knowledge to domain-specific vocabulary).

In compiler theory, the term semantics is also used for *semantic analysis*, which refers to a validation stage that comprises tasks such as scope resolution and type checking after a syntax tree has been produced in a former *syntax analysis* or compiling stage. However, in this work, we use the term semantics exclusively for conceptual or common sense-based interpretations, and thus subsume any information from traditional code analysis tools such as type inference under the term syntax.

The Transformer Architecture

The language models that power semantic technologies—*embedding models* that compute embedding representations of documents and *generative LLMs* that produce plausible text (often referred to as simply LLMs)—are commonly

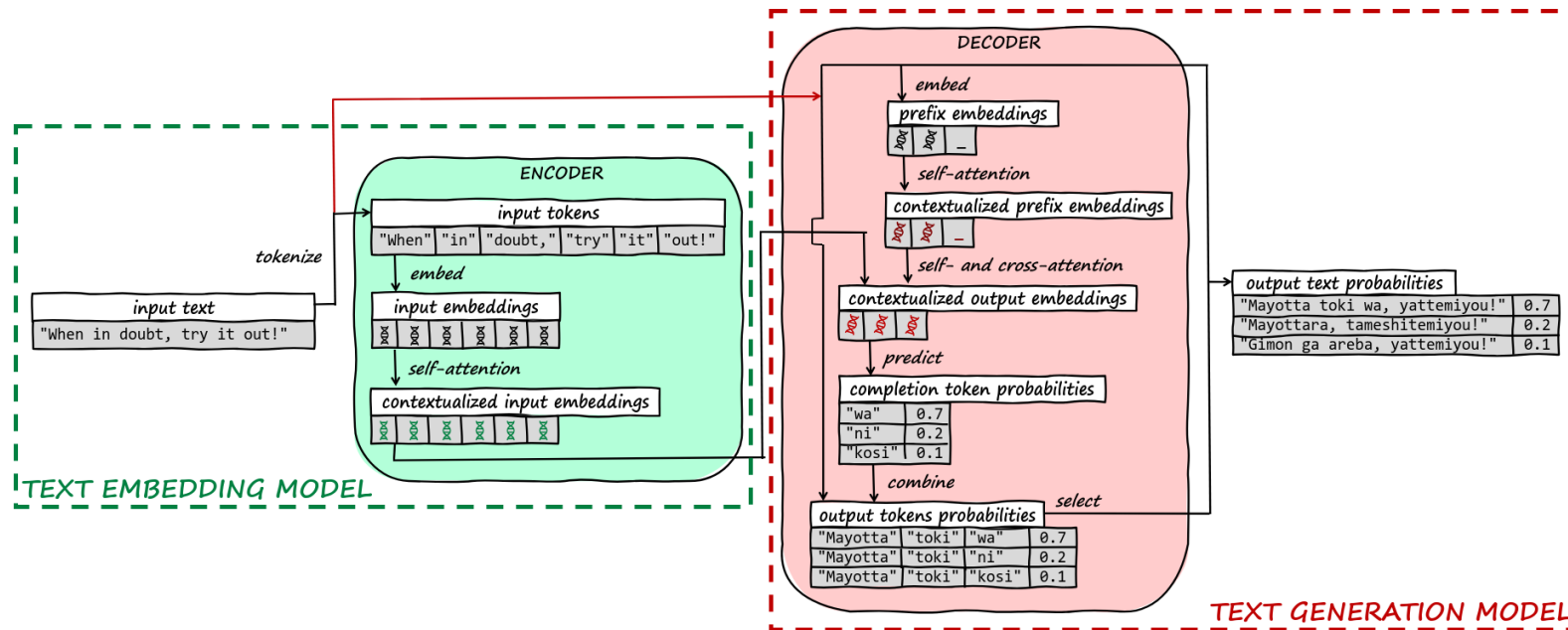


Figure 2.4.: A high-level overview of the transformer architecture, which is used by many traditional language models for translating text from one language (here: English) to another (here: Japanese). The encoder computes embeddings for each input token and contextualizes them with each other. The decoder sequentially generates the output text based on the input embeddings and the preceding output prefix by contextualizing them together and predicting the next token. *Text embedding models* use only the encoder component of a transformer, while many *text generation models* use only the decoder and treat the input text as an output prefix instead (red arrow).

2. Background

based on (different subsets of) the *transformer architecture* [107]⁵ Here, we provide a high-level overview of transformer models. As we intend to introduce a general notion of language models to the exploratory programming community, we focus on their conceptual operating principles and deliberately omit mathematical and technical details.

From a macroscopic viewpoint, a transformer model employs several neural networks to translate a text from an input language into a text in an output language (fig. 2.4). Possible languages include natural languages, programming languages, other formal languages, and (patchwise sequential representations of) multimedia data such as pictures [82] and sounds [84]. The model is usually trained by supervised learning using a large set of input-output pairs. Internally, a full transformer consists of two components (even though modern language models rarely combine both of them): the *encoder* and the *decoder*:

THE ENCODER converts the input text into a sequence of *contextualized token embeddings*. For that, it first splits up the text into a sequence of tokens from a finite alphabet (such as words or parts of words) and embeds these tokens using a pre-trained lookup table. It then contextually enriches these embeddings through multiple layers that comprise a self-attention mechanism and a feed-forward neural network. Through self-attention, the different input tokens are put in relation to each other based on multiple aspects such as syntactical, lexical, and positional relationships. As a result, the contextualized token embeddings describe the semantics of each token in the context of the input text.

Example. In the text “Squeak image”, an encoder would embed the token “Squeak” closer to other tokens describing programming systems such as “Smalltalk” and “Jupyter”. However, in the sentence “Squeak mouse”, the same token would be embedded closer to toy-related tokens such as “noise” and “rubber”.

THE DECODER is invoked multiple times with a sequence of contextualized input token embeddings to *infer* (i.e., generate) a likely output text or a probability distribution of output texts. At each invocation, the transformer predicts the next output token given the contextualized input embeddings and all previously generated output tokens. It contextually enriches the previous output tokens and combines them with the input embeddings

⁵We acknowledge the existence of alternative architectures such as long-short term memory (LSTMs), latent semantic analysis (LSA), convolutional neural networks (CNNs), and hybrid approaches. Still, we restrict our focus in this short introduction to the transformer architecture, which is used predominantly in modern large general-purpose embedding and text generation models [98].

2. Background

through a self- and cross-attentive stack, similar to the encoder. From the contextually enriched output token embeddings, it selects the embedding next to the previously generated token and converts it into a probability distribution of tokens. This process is repeated until the output sequence is terminated with a special end token. Through a *beam search*, the transformer can explore multiple paths of the (usually infinite) probability tree of possible outputs instead of greedily considering only the most likely token at each step.

Text inference in the decoder can be configured through several hyperparameters that modify the probability distributions of output tokens: for example, a *temperature* factor and a *nucleus sampling rate* can alter or truncate the variance of outputs, often associated with the creativity of the model [87]; *frequency penalties* can control the repetitiveness of outputs; or *token biases* can prioritize particular tokens.⁶

In the following, we describe how transformers are used by embedding models for semantic retrieval and by generative LLMs for text completions.

Semantic Retrieval with Embedding Models

To compute a *document embedding* of a text (also referred to as *sentence embedding* or *text embedding*), models such as BERT (bidirectional encoder representations from transformers) use only the encoder component of transformers, which was trained using self-supervised learning, and aggregate the resulting embedding sequence into a single embedding vector [81, 102]. *Semantic retrieval systems* (also referred to as *vector databases* or *vector stores*) compute and index document embeddings for objects such as web pages, files, or source code [89, 92]. This allows them to efficiently perform a *similarity search* in order to recommend semantically related objects to a given object.

Semantic retrieval systems can also offer a more general form of *semantic search* by taking a free-form query from a user, embedding it, and comparing it to the existing document embeddings. Based on the training of the embedding model, this can even yield useful results when query and documents are in different formats (such as questions and answers). For this use case, search quality can be improved by transforming queries and documents into a consistent representation by preprocessing them prior to computing embeddings. For example, it is possible to generate potential questions for documents or to generate fictitious, relevant documents for questions (*hypothetical document*

⁶OpenAI: "Text Generation Models". *OpenAI API Reference*. URL: <http://archive.today/2024.05.30-211619/https://platform.openai.com/docs/guides/text-generation/text-generation-models>.

2. Background

embeddings, HYDE) [83, 95] by using static heuristics or another language model.

Other popular applications of document embeddings include clustering tasks (for unsupervised grouping and anomaly detection) and classification tasks (for supervised grouping, e.g., based on languages or sentiments).

Text Generation with LLMs

Modern generative LLMs such as GPT (generative pre-trained transformer), LLAMA, and PALM use only the decoder component of transformers: instead of encoding and transforming an input text, they treat it as a part of the output text and generate a likely completion of that [99, 101]. They are usually trained using a combination of self-supervised learning with large text corpora and reinforcement learning from human feedback (RLHF) to fine-tune completions [100].

One of the earliest practical applications of LLMs that had a wider impact on the programming community was integrated *semantic code completion* tools such as GitHub Copilot, Tabnine, CodeWhisperer, and others [4, 80], which suggest additions to the code a programmer has typed in an editor.

Beyond simple text completion, LLMs can also be trained to adhere to certain text formats and patterns and follow *instructions* in the prompt, allowing the construction of specialized *agents* with different characteristics:

CONVERSATIONAL AGENTS engage in conversations with human *users* by generating messages (that complete a conversation history) on behalf of a virtual *assistant* [76]. Optionally, *system* messages can be used to provide further instructions to agents. For example, OpenAI’s ChatGPT, Anthropic Claude, and GitHub Copilot Chat allow users to write, edit, review, or summarize text or source code, or discuss a wide range of topics through a chat interface [99].

AUTONOMOUS AGENTS are designed to generate *inner monologues* that mimic structured thinking, resulting in basic capabilities for self-organized problem solving such as step-by-step analysis [111].

Additionally, they can be enabled via pre-training or instructions to access external systems through *function calls*: an LLM emits a special output sequence that requests the invocation of a system function with a set of arguments, a handler executes this invocation and passes back the result to the LLM, and the LLM continues the generation based on the result [85, 96, 111]. For instance, when presented with a mathematical word problem, GPT-4o can break down the task, plan a solution approach, and delegate primitive arithmetic tasks to an external calculator function [99].

2. Background

Building on the concept of autonomous agents, *multi-agent frameworks* such as METAGPT and CHATDEV orchestrate multiple agents that cooperate to solve complex problems such as software development [88, 41].

Being statistical models without an actual understanding of matters in human terms, LLMs suffer from several weaknesses such as limited logical reasoning, not adhering to instructions, and *hallucinations* where false information is generated [99]. To mitigate these problems in parts, several techniques have been established:

FINE-TUNING: Adjust the output style and behavior of models by training a base model on a set of positive examples of texts or conversations [90, 108].

PROMPT ENGINEERING: Adjust the output style and behavior by strategically developing prompts and instructions that are (more) likely to influence models in an intended way [110]. For example, *chain-of-thought* prompting instructs models to explicate thoughts through inner monologue (“think out loud”) or express upfront plans in a certain structure, which has been shown to improve their problem-solving abilities [109].

Unlike fine-tuning, prompt engineering can be applied to any model without retraining, but the prompt must be presented to the LLM for each generation, often affecting its performance [112]⁷ and often several iterations are required to develop effective prompts [110].

RETRIEVAL-AUGMENTED GENERATION (RAG): To mitigate the limited abilities of LLMs to recall rarely seen information, or to provide them with new information, gather excerpts from external sources through traditional algorithms (such as database lookups and full-text searches) and include them in the prompt [92]. This often includes semantic retrieval methods.

For example, the Microsoft Copilot integration in Bing performs one or a few web searches based on the query of the user, feeds the result into a GPT-4o model, and instructs it to answer the query based on the provided information.⁸ Alternatively, agents can proactively call functions to retrieve required information. Thus, RAG addresses the challenge of information recall and moves it from training a model to searching and using information presented in the prompt.

*

Due to their foundation in machine learning methods and training on extensive data, semantic technologies make it possible to process and synthesize

⁷While it is possible to cache the contextualized input embeddings produced in the decoder, only few cloud-based LLMs such as Anthropic Claude currently support this functionality through their API.

⁸Microsoft. 2023-11-21. *How Copilot Works*. URL: <http://archive.today/2024.05.30-235455/https://www.microsoft.com/en-us/bing/do-more-with-ai/how-bing-chat-works>.

2. Background

information based on its context and semantics. We believe that this creates an opportunity to tackle the limitations of traditional exploratory programming systems, which are restricted to technical interfaces at lower abstraction levels.

Example. A programmer seeks help with the task of building a UI prototype from [page 15](#). A traditional tool could support them merely by suggesting lexically related names or systematically testing all permutations or modifications of a code snippet. On the other hand, a semantic tool could *understand*⁹the underlying intent of the programmer, find contextually relevant classes and names, apply them in plausible combinations, or research and fix runtime errors autonomously along the way.

By integrating semantic technologies into exploratory programming systems, we envision providing broader, conceptual, and contextual support for programmers within their exploratory workflow.

Chapter Summary

In the *exploratory programming workflow*, programmers ask questions about software systems and answer them by incrementally decomposing abstractions and conducting experiments. *Exploratory programming systems* support this research process by providing access to the software system through low- to intermediate-level (domain-specific or task-specific) tools.

However, traditional systems cannot understand the context and intentions of programmers, and when programmers translate conceptual questions to technical interfaces manually, they are distracted, overwhelmed, and find only limited answers. *Semantic technologies* provide new AI-based opportunities for processing and synthesizing information and could allow to construct semantic exploratory programming systems to augment the exploratory programming workflow.

⁹We acknowledge the fundamental technical differences between human thinking and machine reasoning. However, contemporary generative LLMs have demonstrated good approximations of human thought processes in several specific domains. Therefore, in this thesis, we use the terms *understanding* and *thinking* in reference to machine agents without consistently putting them in italics or quotation marks. We remain mindful, though, of the limited abilities of artificial agents, particularly in areas such as logical reasoning, common-sense knowledge, and “intuitive” ideas of concepts.

3. Approach

In this chapter, we introduce our model of an *augmented exploratory programming workflow*. This model integrates semantic technologies into exploratory programming systems to support programmers in their research process through more conceptual and contextual interfaces. Second, we present our concept of a *semantic workspace*, which applies our workflow by defining three different kinds of novel *semantic tools* for exploratory programming systems.

3.1. The Augmented Exploratory Programming Workflow

In the traditional exploratory programming workflow, programmers receive limited support from programming systems, which causes semantic distances, information overload, and frequent interruptions. To address these challenges,

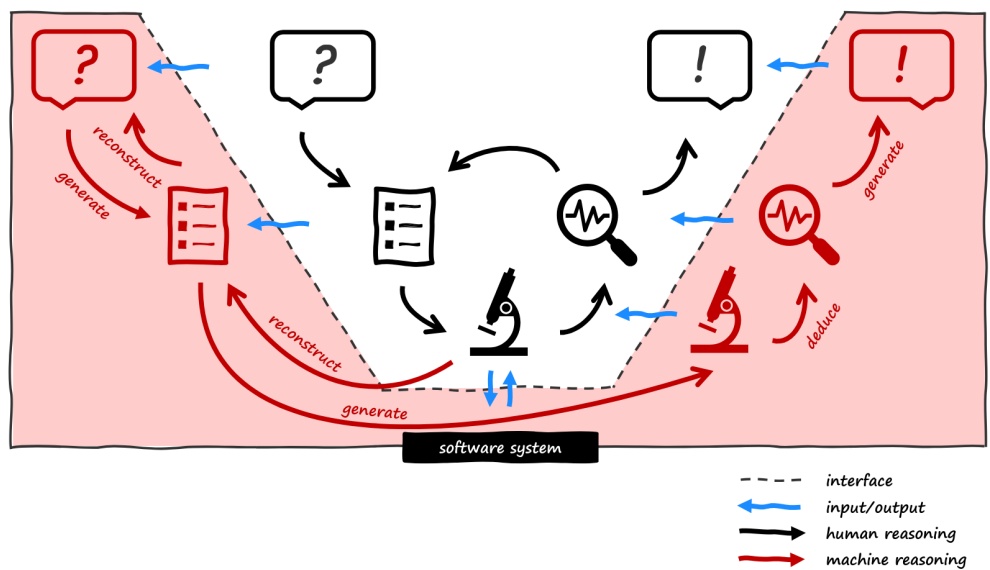


Figure 3.1.: Our model of an augmented exploratory programming workflow. Programmers can exchange conceptual artifacts with a *semantic exploratory programming system* (red) through high-level semantic interfaces. Based on the shared artifacts, the programming system continues the research process and suggests new artifacts to the programmer.

3. Approach

we propose a new *augmented exploratory programming workflow*, which describes collaborations between programmers and systems at higher abstraction levels and introduces the notion of *semantic exploratory programming systems*, which integrate semantic technologies to support programmers in conceptual research steps such as planning and understanding (fig. 3.1).

In our augmented workflow, programmers can provide semantic context from their research process to the exploratory programming system, such as questions, plans, and previous experiments. The system builds on this context to “think along” and continue the research process on its own: it attempts to conduct next likely steps such as planning and executing experiments, deducing results, or answering questions and shares its work with the programmer at different granularities. For example, a programmer could provide a high-level question to the system and receive a list of suggested experiments to execute; or they could perform their own experiments and receive an automated summary of deduced results from the system. Thus, programmers can either *delegate* tasks to the system to avoid interruptions, or they can *cooperate* with it to benefit from semantic technologies and *augment* their workflow with additional context.

To support this workflow, we propose *semantic exploratory programming systems*: these systems build upon traditional exploratory programming systems and connect them with semantic technologies to gain access to the semantic context at higher abstraction levels of the research process. For example, this connection allows exploratory programming systems to interpret questions and plans, contextualize and analyze experiments, or summarize results and answer questions. Using this context, semantic exploratory programming systems can replicate the previous research steps of programmers and suggest possible next steps.

To access and contribute to the exploratory research process of programmers, semantic exploratory programming systems require new *semantic interfaces*, through which programmers can provide contextual artifacts as inputs and retrieve semantic suggestions and answers as outputs. Concretely, we propose three types of semantic interface mechanisms:

ANTICIPATED EXPERIMENTS: The semantic exploratory programming system observes the *experiments* that the programmer executes through traditional interfaces. Based on these observations, it attempts to reconstruct their underlying plans and uses these plans to anticipate and suggest further experiments to the programmer.

SEMANTIC INPUTS: The semantic exploratory programming system reinterprets existing or introduces new interfaces, through which programmers can express their current *plans*. For example, this involves reading quick notes of programmers (such as to-do lists kept in a workspace), observing the setup

3. Approach

of experiments before the programmer executes them, or—hypothetically—even listening to programmers thinking out loud during their work. Based on these inputs, the system can develop a more precise understanding of the programmer’s plans and provide more relevant suggestions to them.

Furthermore, the system can also attempt to reconstruct the overarching *question* of the programmer based on the provided plans. This serves as a base for anticipating further plans and developing suggestions for them even before programmers express these plans to the system.

Finally, the system can offer new interfaces, through which programmers can explicitly express high-level questions. The system can use these to refine existing and new plans and provide more contextualized suggestions.

SEMANTIC OUTPUTS: Based on conceptualized ideas and questions, the semantic exploratory programming system can provide suggestions to programmers through new output interfaces at different abstraction levels. It can suggest low-level *experiments* or execute them on its own and automatically deduce and summarize *results*. If the system knows the original question of the programmer, it can also consolidate and contextualize results to provide a direct *answer* to this question.

Internally, semantic technologies are used to reconstruct plans and questions of programmers, make suggestions, and deduce results and answers. [Chapter 5](#) describes how text embeddings and semantic retrieval are employed to anticipate plans and experiments, and [chapter 6](#) explains how systems can use generative LLMs to interpret questions, generate experiments, and deduce answers.

3.2. The Semantic Workspace

To study our concept of an augmented exploratory programming workflow, we describe the *semantic workspace* as a conceptual framework for semantic exploratory programming systems that contain different *semantic exploratory programming tools*. Within this framework, we investigate different manifestations of the augmented exploratory programming workflow through semantic tools that support programmers at different levels of abstraction in their research process and use different types of semantic interfaces.

All of these manifestations can be arranged along a spectrum based on their degree of support between two extremes ([fig. 3.2](#)):

AUGMENTATION: At one extreme of the spectrum are tools that operate at a lower level of abstraction and provide narrow support to programmers. Low-level semantic tools primarily employ technical interfaces and artifacts

3. Approach

such as experiments and *augment* the research process of programmers by providing suggestions for smaller parts of their workflow, such as further experiments.

AUTOMATION: At the other extreme are tools that provide wide-ranging, more conceptual semantic support. In contrast to low-level semantic tools, they often employ new semantic interfaces to exchange conceptual artifacts with programmers. Rather than augmenting the exploratory programming workflow, they tend to *automate* parts of it and provide programmers with aggregated results and answers instead of low-level suggestions.

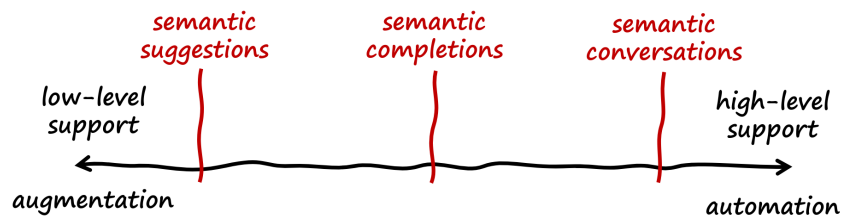


Figure 3.2.: The support spectrum of different tools in the semantic workspace. Low-level semantic tools collect context from tracking the actions of programmers in the programming system and *augment* their exploration through new suggestions. High-level semantic tools retrieve explicit conceptual descriptions from programmers and *automate* a larger part of the research process.

Orthogonally to their degree of support, semantic tools can exhibit reactive or proactive interaction mechanisms: reactive tools expect an explicit question or invocation by the programmer, while proactive tools observe the steps of programmers in the background and make suggestions before a programmer explicitly requests support. Similar to the degree of support, semantic tools can combine characteristics of both interaction mechanisms.

The choice between augmenting and automating, or between reactive and proactive semantic tools, relates to the current objectives of programmers and has different implications for their overall programming experience, trust, and learning curve, which we will discuss in [chapter 9](#).

In the semantic workspace, we propose three concepts for novel semantic tools that can be added on top of traditional exploratory programming systems and that together provide different means for exploratory programmers to augment and partially automate their workflow:

SEMANTIC SUGGESTIONS anticipate the plans of programmers by continuously monitoring their experimental activities and proactively suggesting further experiments as well as optional summaries.

3. Approach

SEMANTIC COMPLETIONS monitor the planning activities of programmers as they draft code or text in editors, automatically generate and conduct experiments, and proactively provide contextualized suggestions such as code snippets in the editor.

SEMANTIC CONVERSATIONS provide an interface through which programmers can express conceptual questions about objects (such as domain objects or classes) in natural language and retrieve answers at the same abstraction level. Internally, an exploratory programming agent conducts the necessary plans, experiments, and deductions to generate answers.

In the following, we describe each semantic tool concept in detail.

3.2.1. Semantic Suggestions

The approach of *semantic suggestions* is to augment the research process of exploratory programmers by providing them with possible experiments based on their previous experimental activities. For this, the semantic workspace tracks the experiments that programmers execute through the interfaces of traditional exploratory programming systems such as do-its, code browsers, and method finders. Based on these observed experiments, it attempts to reconstruct the plans of programmers, such as an intention to understand the (scattered) implementation of a concept or to develop a new prototype for a user interface. Using this reconstructed plan, the semantic workspace then generates further potentially relevant experiments and suggests them to the programmer.

Suggestions are proactively generated and presented in a list-like interface, such as a dock that is placed next to other tools in the system through which programmers conduct experiments (fig. 3.3).

Semantic suggestions exploit the ability of semantic technologies to process information at a larger scale than human beings, albeit with reduced precision and recall. For this reason, the number of generated suggestions is usually in the two- to three-digit range. This necessitates a structured and ranked presentation of suggestions, which is achieved by grouping them based on their type (such as classes versus methods) and ranking them based on their relevance, similarity, and diversity (see section 5.3).

Optionally, semantic suggestions can include summaries and deduced results of all experiments. For example, the central aspects of a complex implementation or lengthy documentation could be briefly summarized in the context of the anticipated plans of programmers as a natural-language synopsis.

Semantic suggestions can be applied for different types of plans and experimental activities, including but not limited to the following use cases:

3. Approach

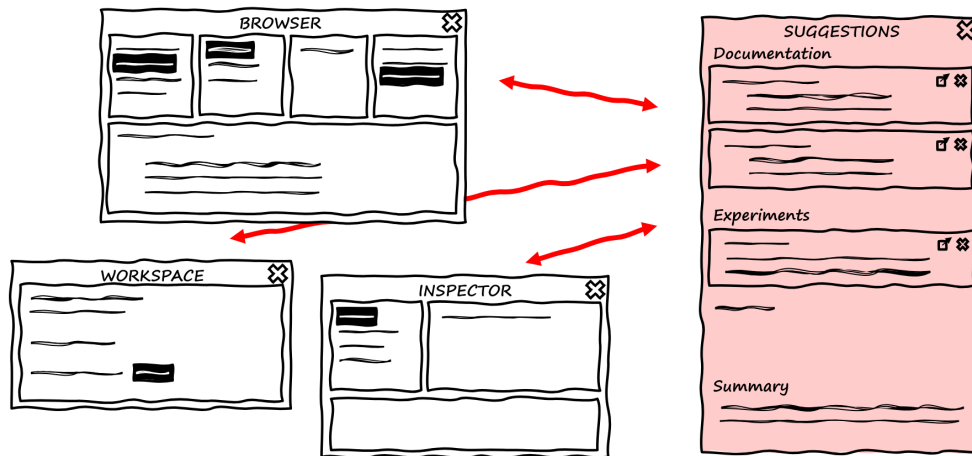


Figure 3.3.: Our concept of a *semantic suggestions* tool in the semantic workspace. The augmented programming system tracks the experimental actions of the programmer through traditional programming tools (**gray**), anticipates the possible next steps of the programmer, and provides them with a list of suggestions for further experiments as well as an optional summary of their results (**red**).

Code browsing To understand the implementation of a concept, a programmer is reading one or multiple methods through traditional code browsers. The semantic workspace anticipates this intention and thus suggests further methods and code artifacts that the programmer might want to browse next. This includes:

- SENDERS of the currently viewed methods: methods that (possibly) invoke the current methods;
- IMPLEMENTORS of messages that the current methods send, displayed with their comment and implementation;
- CLASSES that the current methods reference, displayed with their comment and definition;
- SIMILAR METHODS, such as methods that pursue the same goal or implement common strategies.

Beyond code artifacts, the system could also suggest other types of semantic suggestions in this context, such as relevant excerpts from related documentation artifacts, design documents, and specifications.

Depending on the quantity and complexity of related artifacts, the system might also summarize all suggestions; for example, by describing the implementation of a scattered concept or shortening and paraphrasing documentation artifacts.

3. Approach

Prototyping While programmers write and repeatedly save a new method as part of creating a prototype, the semantic workspace anticipates the type and shape of the desired prototype. To support this activity, it provides the programmer with potentially relevant suggestions for related resources:

SIMILAR METHODS, such as methods that pursue the same goal or implement similar strategies to the current unfinished method;

CORRELATED METHODS AND CLASSES that are used by those similar methods and that the programmer thus might consider using in the current prototype method as well (resembling the concept of collaborative filtering, i.e., “programmers who have sent this message have also sent that message”), displayed with their comment and usage samples;

DOCUMENTATION ARTIFACTS that describe the concept and intended usage of present and suggested classes and methods.

Debugging Semantic suggestions can support programmers who are experiencing an error while using the software system and are investigating it in a debugger to understand its causes and solve it. To this end, the semantic workspace automatically searches different sources to provide several types of suggestions:

BUG REPORTS on bug trackers or other communication platforms that refer to the same error message;

DOCUMENTATION ARTIFACTS that describe causes or solutions for this error;

EXCEPTION HANDLERS for this and similar errors in the software system.

Additionally, the system might also summarize the results to extract and suggest particular debugging and solution strategies (such as checking the state of an object, running a script to repair it, or inserting a missing exception handler).

*

Thus, semantic suggestions support exploratory programmers in their research process by providing them with further possible experiments and automated deductions. This not only accelerates specific research activities but also augments the insights of programmers, as they can gain inspiration from suggestions that might not have occurred to them or would have been associated with too high a manual research cost.

3. Approach

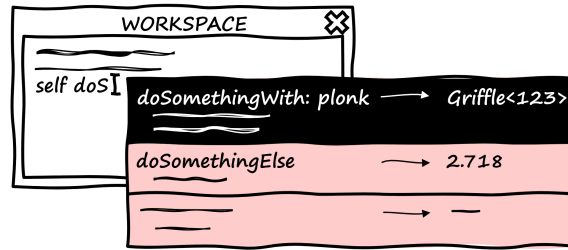


Figure 3.4.: Our concept of a *semantic completions* tool in the semantic workspace. While programmers gradually express their plan to the system, e.g., by drafting a script in a traditional workspace (**gray**), the augmented programming system reconstructs their underlying intentions and suggests contextualized experiments as completions of their existing inputs (**red**).

3.2.2. Semantic Completions

Similarly to semantic suggestions, *semantic completions* observe the activity of programmers in the system and proactively provide them with further contextually relevant suggestions. However, semantic completions support programmers at higher levels of abstraction in their research process: instead of tracking manual experiments of programmers and reconstructing their possible plans from these, semantic completions exploit existing interfaces of exploratory programming systems through which programmers can express their plans, and derive an understanding of these plans directly. Concretely, these interfaces involve different types of editors through which programmers create and edit text, code, or domain-specific objects (fig. 3.4). The semantic workspace does not wait until the programmer finishes their input and submits it to the system but tracks their editing activity continuously (e.g., as they type or click) to follow along with their plans.

Feedback from semantic completions is provided to programmers through conceptual, highly contextualized results instead of single experiment suggestions. This manifests as a list of completions, which are integrated into the editor interface and represent possible continuations of the preliminary plan that programmers have typed into the editor.

To generate these completions, the semantic workspace internally predicts, executes, and evaluates possible experiments of programmers based on their observed plans, similarly to semantic suggestions. After that, results are filtered, aggregated, and transformed into the input context of the programmer.

Semantic completions can be applied to different types of planning activities that exploratory programmers conduct as part of their research process, for example:

3. Approach

Prototyping While programmers write or edit the code of a script or method (e.g., through a textual or visual editor), the system observes their activity and derives their plans. It automatically researches similar methods, correlated class and method names, and relevant documentation artifacts. Based on this information, it synthesizes possible code snippets (that may range from single expressions to entire methods) and suggests them as contextually adjusted completions to the programmer in their editor.

Optionally, these contextualized suggestions can also be dynamically enriched. For example, the system can present a completed code expression together with the result of evaluating it (using available context such as call stacks, Babylonian examples, or symbolic execution, see [page 9](#)) or display the effect of a completed method, such as a preview of a graphical prototype.

Human communication Another frequent activity of programmers is writing text to communicate with other programmers, such as documentation artifacts, bug reports, and commit messages. This can be considered another type of plans and experiments that programmers conduct to express their intentions to the system and benefit from the outputs of the system (e.g., through answers to their questions from other programmers or through reminders to their “future selves” about their original intentions)¹

Semantic completions can assist programmers with writing such natural-language text. To this end, they can automatically browse and suggest relevant classes, methods, or other artifacts such as bug reports and commits; generate example experiments; and complete the sentences started by programmers to fill in the collected information. For example, when a programmer creates a bug report, the system could automatically summarize prior notes and experiments from other workspaces into a consolidated text and insert a test to reproduce the bug.

*

Thus, semantic completions can support programmers in creating and realizing plans by directly following their intentions, generating and conducting likely experiments, and contributing results to the programmers’ planning activity in a contextualized form. This allows programmers to interact with the programming system at a more conceptual level, accelerate their workflow for trivial tasks, and explore different solutions based on an extended, diverse amount of information.

¹Arguably, this perspective can also be applied to source code which “must be written for people to read, and only incidentally for machines to execute” [1, p. xxii].

3. Approach

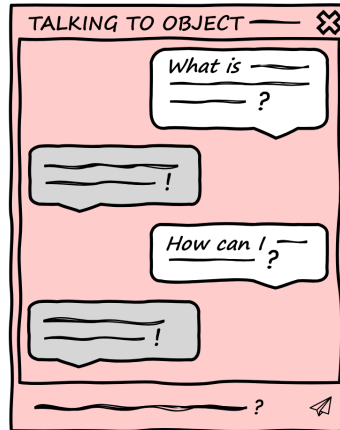


Figure 3.5.: Our concept of a *semantic conversations* tool in the semantic workspace. Programmers express their abstract, contextual questions to the augmented programming system by interacting with an object in the software system through messages in natural language. Internally, the programming system conducts all steps of the required research process for answering the questions autonomously and returns a conceptual, natural-language answer to the programmer.

3.2.3. Semantic Conversations

Semantic conversations support exploratory programmers at high abstraction levels in their research process. For this, a programmer can ask a conceptual question about the software system, and the semantic workspace will return an answer to that question in a similar conceptual, natural-language form (fig. 3.5). The programmer can ask follow-up questions based on the system's answer, and the system may also ask questions back to the programmer to clarify their intentions, resulting in a conversational interaction style.

Internally, the system executes an entire instance of the (potentially nested) research process on its own: it generates a plan to answer the question, generates, executes, and analyzes one or several experiments, repeats as necessary, and finally creates an answer to the question based on the gathered results.

Following the object-oriented philosophy of many exploratory programming systems such as Smalltalk systems, we define one reference object from the system for each semantic conversation. Thus, just like programmers can send messages to objects to inspect and interact with them, they can also ask them natural-language questions through semantic conversations.

3. Approach

Example. A programmer inspects an order object from a shopping system and wants to know when the order was created. To find this out, they ask the object “when were you created” through a conversational interface. In response, the system internally conducts multiple experiments to explore the inner structure and protocols of the order’s class, inspect the internal state of the order object or send messages to it, and run further scripts to convert a retrieved value into a proper format. Finally, the system answers the question of the programmer with an answer like: “I was created on March 14th this year.”

Reference objects can be arbitrary objects from the system: domain objects, code objects such as classes and methods, or derived artifacts of software systems such as test results, program traces, and views from other tools.

Thus, semantic conversations provide an abstract, reactive interface through which programmers can express questions about objects and communicate with the system in natural language. This allows them to avoid context switches and distractions and handle overarching research processes more efficiently. At the same time, the system is able to process more information than the programmer within the same time, creating an opportunity to provide a more extensive overview of a part of the software system within a single conversation.

Chapter Summary

We have proposed the *augmented exploratory programming workflow*, in which programmers collaborate on high-level steps of their research process with *semantic exploratory programming systems*. These systems offer *semantic interfaces* for exchanging conceptual artifacts with programmers and can *anticipate* and *augment* their workflow. Our *semantic workspace* provides three tool concepts for such systems at different levels of support: *semantic suggestions*, *semantic completions*, and *semantic conversations*.

4. The Semantic Exploration Kernel

In this chapter, we describe the design of our *semantic exploration kernel*, which connects the interfaces of the different semantic workspace tools to a traditional exploratory programming system and to semantic technologies. The semantic exploration kernel consists of two central components: a *suggestion engine* for processing context from the programming system and developing suggestions, and an *exploratory programming agent* for natural-language communication, machine reasoning, and autonomous experimentation. In the following, we elaborate on our architecture of the semantic exploration kernel and describe the general operating principles of the suggestion engine and the exploratory programming agent.

4.1. Architecture

Figure 4.1 displays the high-level architecture of the semantic exploration kernel and its environment. The overarching semantic exploratory programming system can be divided into three layers: the *semantic workspace* for (graphical) programming tooling, the *semantic exploration kernel* for the domain logic of the system, and the *infrastructure layer* for required packages and integrations.

The semantic workspace provides different semantic tools, through which programmers can interact with the system (section 3.2). The semantic exploration kernel consists of two components for processing the semantic context of the programming session and augmenting it: the suggestion engine and the exploratory programming agent.

THE SUGGESTION ENGINE captures the implicit concept of programmers from their previous experiments, reconstructs their plans and intentions, and creates new suggestions. For this, it defines a *blackboard framework*, which manages various *artifacts* such as methods, classes, and scripts and orchestrates *strategies* for suggesting new artifacts based on existing ones (section 4.2). Next to the generic framework, the suggestion engine also provides several types of artifacts and strategies that employ semantic retrieval methods for searching objects, generating experiments, and structuring and ranking results (chapter 5).

THE EXPLORATORY PROGRAMMING AGENT receives explicit questions and answers them by autonomously conducting exploratory research processes.

4. The Semantic Exploration Kernel

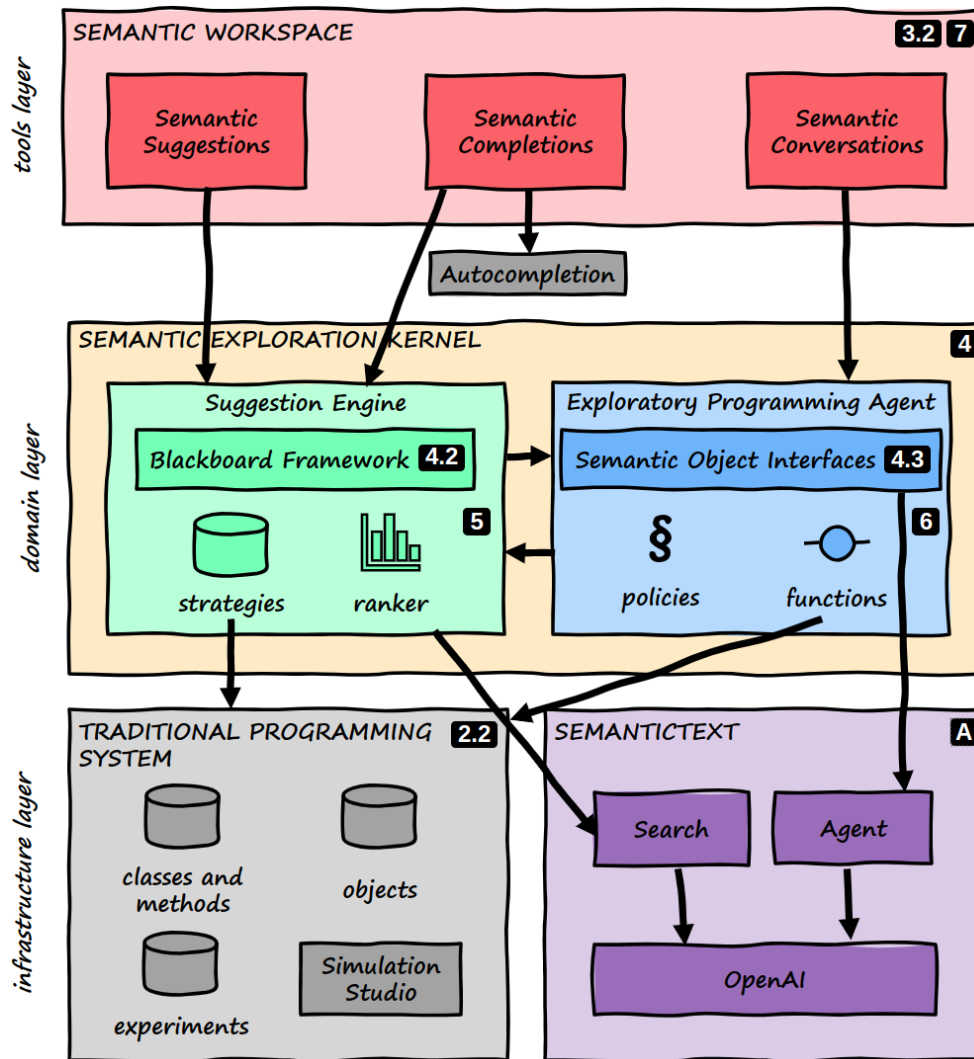


Figure 4.1.: The high-level architecture of the *semantic exploration kernel* and its environment. The architecture is structured in three layers for semantic tools, domain concepts, and underlying infrastructure.

Legend. New components are colorful and existing components are gray. Arrows indicate dependencies from a higher-level component to a lower-level component. **Numbers** reference the chapter, section, or appendix in this thesis that further describes a component.

4. The Semantic Exploration Kernel

It connects to its environment through *semantic object interfaces*, which allow programmers to ask semantic questions to objects in the system (section 4.3). Internally, it employs a generative LLM to support natural-language conversations and machine reasoning. The agent is configured through a set of *policies*, which define its behavior, and a set of *system functions*, through which the agent can interact with objects in the system (chapter 6).

The suggestion engine and the exploratory programming agent can interact with each other: particular strategies that require reasoning about other artifacts or code generation invoke the exploratory programming agent. In the other direction, when the agent needs a broader overview of system parts, it will employ the suggestion engine to retrieve relevant artifacts.

Both the semantic workspace and the semantic exploration kernel require a traditional exploratory programming system (chapter 7). The semantic workspace integrates semantic interfaces for its tools into the existing programming system, for example, through a completion menu in code editors or a conversational interface in several exploration tools. Similarly, it also hooks into the existing user interface to observe experiments through artifacts such as opened tabs and windows. The exploration kernel interacts with the traditional system to access classes, methods, and objects for retrieving suggestions or executing experiments.

In our implementation, the semantic workspace and the exploration kernel also depend on a couple of other packages:

SEMANTICTEXT: The exploration kernel uses our SEMANTICTEXT framework¹, which provides access to semantic technologies for retrieving documents, generating text with conversational LLMs, and implementing specialized agents (appendix A). This framework also provides a graphical user interface component for conversational agents, which is employed by the semantic workspace for semantic conversations.

SIMULATIONSTUDIO: Furthermore, the exploration kernel uses our SIMULATIONSTUDIO framework for instrumented code interpretation² to construct a dynamic call graph about suggested methods (*method harvesting*, page 50) and execute experiments from the agent in an isolated sandbox (section 6.2).

AUTOCOMPLETION: Finally, we use and extend the AUTOCOMPLETION package³ to integrate semantic completions into the programming system (section 7.2).

¹<https://github.com/hpi-swa-lab/Squeak-SemanticText>

²<https://github.com/LinqLover/SimulationStudio>

³<https://github.com/LeonMatthes/Autocompletion>

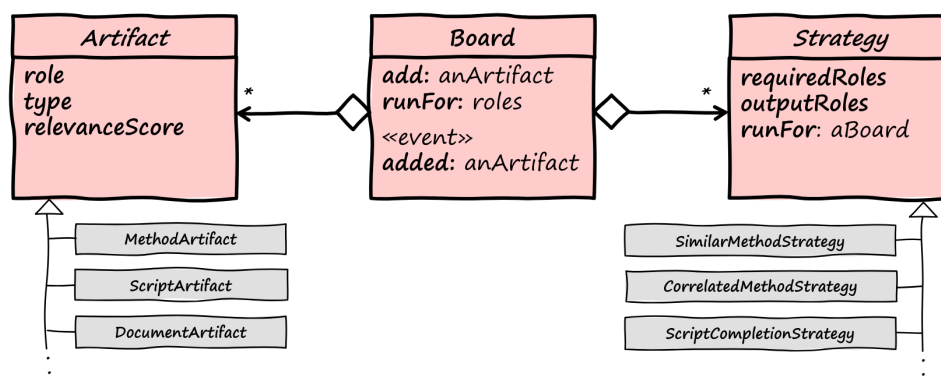


Figure 4.2.: The general object model of our blackboard framework for creating suggestions (UML class diagram with Smalltalk-styled method signatures). A central *board* maintains many *artifacts* of different types and roles, and it has access to *strategies*, which consume existing artifacts to produce new artifacts. When asked for suggestion artifacts of certain roles, the board identifies and schedules all required strategies and returns the new suggestions to the requestor.

In the following sections, we introduce our blackboard framework for the suggestion engine and semantic object interfaces for the exploratory programming agent.

4.2. A Blackboard Framework for Managing Suggestions

When the suggestion engine is invoked, it first captures the implicit context of programmers as *artifacts*—for example, currently viewed methods, recent experiments, or notes. It then executes different *strategies* to suggest further artifacts based on the former. For example, given a currently viewed method, different strategies could suggest senders of this method, similar methods, or related documentation artifacts. To anticipate larger parts of the programmer’s research process, suggestions are not limited to initial input artifacts but can also be derived from prior suggestions. In our example, a fourth strategy could extract and suggest popular message sends from similar methods.

These requirements create a high complexity in the suggestion engine, which has to manage different types of artifacts, strategies, and a convoluted dataflow. To handle this complexity and maintain flexibility in the design of new artifacts and strategies, we define a *blackboard framework*, which organizes artifacts and orchestrates strategies.

The framework defines a central board on which different artifacts can be arranged. Each artifact is specified with a *role* (such as input artifact, similar

4. The Semantic Exploration Kernel

artifact, or method sender) and a *type* (such as class, method, or documentation). Additionally, the board contains a set of strategies, each of which is able to consume given artifacts and produce more artifacts. Each strategy can specify a set of *required roles* for the artifacts it consumes as well as a set of *output roles* for the artifacts it produces. Figure 4.2 displays our object model of the blackboard framework.

After the board has been configured with a concrete set of input artifacts and strategies, it can be invoked to provide suggestions of certain roles. Once invoked, the board determines all available strategies that can produce the requested artifact roles and schedules them. If any strategy defines requirements, the board first checks whether they are met by the current set of artifacts, or otherwise attempts to determine and schedule further strategies that produce artifacts of the required role. This process continues recursively until all possible strategies that are directly or indirectly required to fulfill the request have been scheduled. All scheduled strategies are executed concurrently as soon as their requirements have been resolved, and each running strategy adds new artifacts to the board. Finally, the requestor can access or stream the resulting artifacts from the board.

The board framework also provides strategies for organizing artifacts: for example, identical or equivalent artifacts that have been produced by multiple strategies can be merged, artifacts can be grouped based on their role, type, or metadata, and they can be sorted based on ranks provided by prior strategies.

To support debugging and observability, strategies can also be executed synchronously, and a provenance mechanism remembers the original strategy and its input artifacts for every produced artifact.

Example. A programmer has drafted a new method and a short list of to-do notes in their programming system. To provide them with further possibly relevant message sends for their method draft, the semantic workspace invokes the suggestion engine with both artifacts and requests suggestions of the role “correlated message” (fig. 4.3).

Once invoked, the suggestion engine identifies two strategies that can suggest correlated messages: a *correlated selectors* strategy and a *method harvester* strategy (see page 50). Both strategies require artifacts of the role “similar method”, which are currently not present on the board. For this reason, the suggestion engine finds two other strategies that can produce similar method artifacts: a *semantic method search* strategy and a *TF-IDF selector search* strategy. Both these search strategies require artifacts of the role “input”, thus their requirements are met and the engine schedules them.

4. The Semantic Exploration Kernel

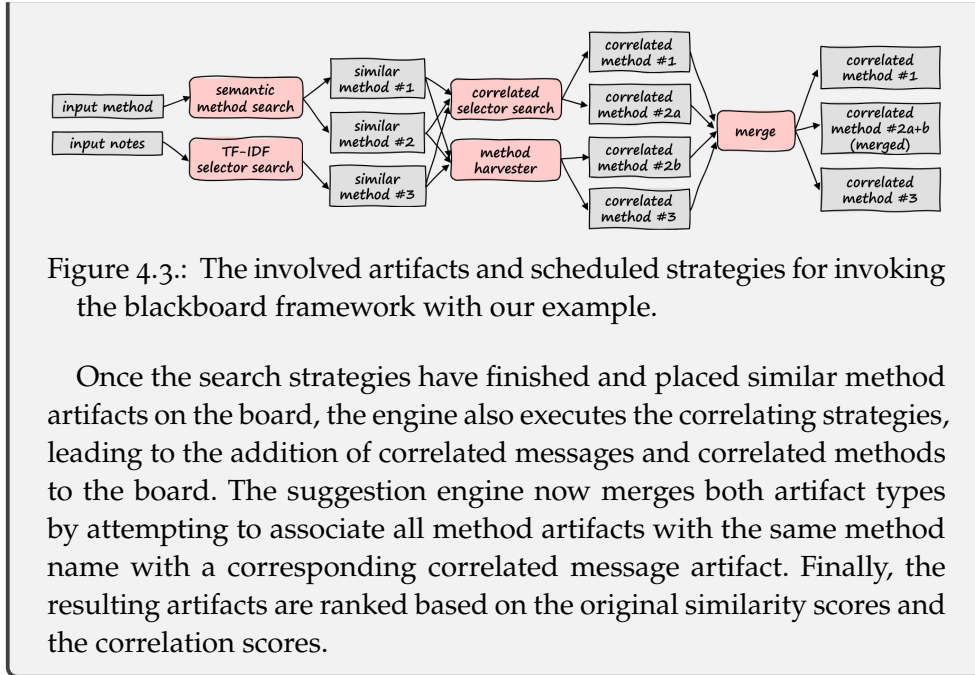


Figure 4.3.: The involved artifacts and scheduled strategies for invoking the blackboard framework with our example.

Once the search strategies have finished and placed similar method artifacts on the board, the engine also executes the correlating strategies, leading to the addition of correlated messages and correlated methods to the board. The suggestion engine now merges both artifact types by attempting to associate all method artifacts with the same method name with a corresponding correlated message artifact. Finally, the resulting artifacts are ranked based on the original similarity scores and the correlation scores.

Thus, our blackboard framework shares characteristics from two existing approaches: *blackboard systems* and *data orchestration pipelines*. First, it resembles the *blackboard architecture pattern* [8, p. 71ff.] and *blackboard systems* from early expert systems [86], where different *knowledge sources* successively populate a *blackboard* orchestrated by a *control shell*. Second, it is related to *data orchestration pipelines*, which model and execute a directed acyclic graph of *tasks* [63].

However, our blackboard framework follows a more structured approach than traditional blackboard systems in that all strategies define static output roles, allowing to predict the dataflow in advance and schedule strategies in a top-down manner (as opposed to the bottom-up method of blackboard systems). At the same time, the blackboard framework offers greater flexibility than data orchestration pipelines, which typically model direct dependencies between tasks rather than data roles, and thus allows for the implicit choice and combination of different strategies.

4.3. Semantic Object Interfaces for Exploratory Programming

To delegate parts of the research process to the exploratory programming system, we construct an *exploratory programming agent*, which uses a generative LLM for planning, generating experiments, and communicating with the programmer in natural language. To connect the agent with the programming system, we propose the design of *semantic object interfaces*, which describes the

4. The Semantic Exploration Kernel

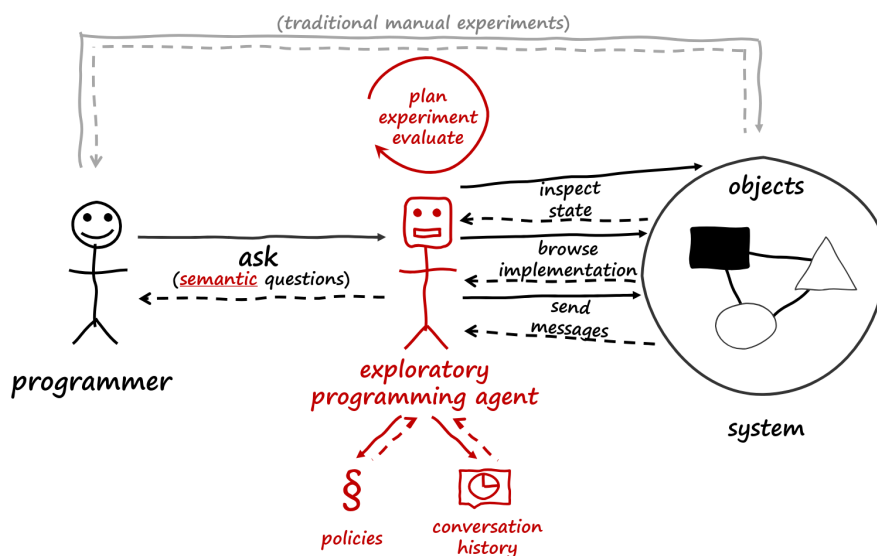


Figure 4.4.: Our approach of semantic object interfaces for semantic exploratory programming systems. The programmer expresses high-level, contextual, and often natural-language questions about an object to the interface and receives answers on the same abstraction level. Internally, an *exploratory programming agent* (red) translates these questions and interacts with the system to perform low-level experiments.

communication between the programmer, the agent, and the system [70]. This design follows the object-oriented paradigm of many programming systems in that semantic object interfaces focus on a single object from the running system. Thus, semantic object interfaces resemble traditional message sending to objects, and programmers can interact with objects via the agent by means of natural-language and semantic questions.

As part of this framework, the agent takes high-level and context-dependent *semantic* questions from the programmer and translates them into low-level technical experiments to answer the questions (fig. 4.4). Our framework comprises three fundamental actors:

THE PROGRAMMER conducts a larger exploratory research process, from which different questions will arise. The programmer asks such questions about an object in a system and expects answers on a similar level of abstraction. These questions are *semantic* and conceptual, meaning that they are expressed in the programmer’s mental model and vocabulary, typically have an informal or natural-language style, and often depend on the context of previous questions and answers.

THE OBJECT is any object in the system, for example a particular domain object, a code object (such as a class or method), or an artifact from the

4. The Semantic Exploration Kernel

programming system (such as a call graph, a benchmark, or a program trace). It is part of an *object graph*, which embeds it into a larger context of the system. It is also linked to an *implementation*, which describes the object's behavior through further objects and may be specified using code, tests, or other forms such as contracts.

Objects can be accessed either by sending them *messages* or through *reflection interfaces* that expose their internal state, implementation, or location in the object graph.

THE EXPLORATORY PROGRAMMING AGENT is an intelligent mediator between the programmer and the object in the system. It automatically translates conceptual questions from the programmer into interactions with the system and translates the results of these interactions back into answers for the programmer.

Internally, the agent conducts research processes autonomously: it interprets the programmer's question, develops a plan, designs and conducts experiments, evaluates their results, and repeats as necessary (until all required information has been collected) before delivering a reasoned answer to the questions and returning it to the programmer.

The agent uses two resources: a set of *policies* and a *conversation history*. *Policies* define abstract agent behavior, such as the types and frequency of experiments and the format of answers. The *conversation history* consists of past communications with the programmer and experiments from the current conversation. It serves as a context for handling subsequent requests. Thus, the programmer does not need to repeatedly explicate their intentions in every question but grows a shared vocabulary and knowledge with the agent as the conversation evolves.

Answering Questions through Automated Experiments

Our framework allows programmers to ask arbitrary questions about objects in a system, which can reference two different aspects:

FUNCTIONAL QUESTIONS (or "what" questions) refer to the state of objects and the actual things in the domain they represent. They typically constitute inquiries that are or could be covered by regular (analytical) system features or objects' behavior.

For example, in a sales system, typical functional questions could be "How many customers are there?", "Which product in this category has generated the highest profit in the last quarter?", or "What is the age distribution of weekend customers?".

EPISTEMIC QUESTIONS (or "how" questions) refer to the behavior of objects, domain concepts, and their implementation. Programmers ask these questions

4. The Semantic Exploration Kernel

to explore the capabilities of a system, understand the technical foundations, or ideate and prototype new applications.

For example, in a sales system, epistemic questions could include “What information do we store about customers?”, “How is the tagging system for products modeled?”, or “How can we analyze the shopping behavior of customers?”.

Note that given the uniform object model of many programming systems, epistemic questions about a domain object equal functional questions about a code object from the implementation of the domain; however, the former perspective provides additional context about the system through a concrete example.

To answer both functional and epistemic questions, the agent automatically conducts experiments by utilizing three types of interfaces that most existing programming systems already offer:

STATE INSPECTION allows the internal information of objects to be explored.

For example, the agent can look up a variable or enumerate all properties of an object.

IMPLEMENTATION BROWSING explores the specified behavior of objects through their *protocols*, *implementation*, and *documentation*. *Protocols* refer to the set of messages an object understands. *Implementation* includes the source code of objects or classes, but also their integration within the global system through call graphs or related concepts (e.g., senders or program traces) for understanding the usage of objects by example. *Documentation* can be provided through comments, examples, or alternative system descriptions.

Following the singularity of objects and meta objects noted above, implementation browsing of domain objects equals state inspection of their classes, but is associated with a different connotation by adhering to the context of the original object. For example, using symbolic execution methods, both the state and effective behavior of objects can be explored at once.

MESSAGE SENDING constitutes regular communication with objects to activate their behavior. As opposed to state inspection and implementation browsing, normal messages can be sent without requiring reflective capabilities.⁴

Note that the agent generally does not require any manual preparation for specific systems and packages from domain experts or programmers but will learn about the system on its own. Thus, even to answer simple func-

⁴From our classification, we exclude special reflective messages such as #instVarNamed: in Smalltalk as well as restricted visibility of messages such as the private access modifiers in Java.

4. The Semantic Exploration Kernel

tional questions, the agent will internally ask and answer epistemic questions to understand the system and domain concepts and preserve the collected knowledge in its internal conversation history. Analogously, every message sent to a previously unknown system is preceded by browsing its implementation, through which the agent discovers the relevant protocols and messages to use.

Example. A programmer asks “Which product has generated the highest profit?” about a selected shop object. In response, the agent first executes a series of experiments by browsing the shop’s implementation to explore several messages, classes, and their documentation related to the concepts “product” and “profit” to understand what these concepts mean and how they are represented in the system.

After identifying relevant messages such as `Shop»orders`, `Order»productItems`, and `Product»price`, it plans how to combine this information to compute the most profitable product and runs a script that queries the system for this information as another experiment.

Finally, the agent evaluates the results of this experiment and returns a summarized answer in natural language to the programmer. The question was answered and the programmer can continue by asking another question to the agent or performing another exploratory activity.

Integrating Semantic Object Interfaces into Exploratory Programming Systems

To support immediate access to semantic object interfaces from within exploratory programming systems, we aim for tight integration with traditional tools. For this, we identify two primary interfaces in such systems through which programmers explore objects: *object inspection tools* and *message sending through scripts*. We propose to extend these interfaces with semantic capabilities: object inspection tools with a *conversation mode* and message sending with a language extension for *semantic messaging*.

A conversation mode for object inspection tools Inspection tools allow programmers to explore and manipulate the internal state of objects through a list of variables or properties. We propose a new *conversation mode* for inspection tools, which allows programmers to ask semantic questions about an object (fig. 4.5). Thus, in addition to usual technical state inspection, they can also chat with an object through the exploratory programming agent.

Through the chat interface, programmers can express questions in natural language without needing to know the vocabulary and protocols of the object’s

4. The Semantic Exploration Kernel

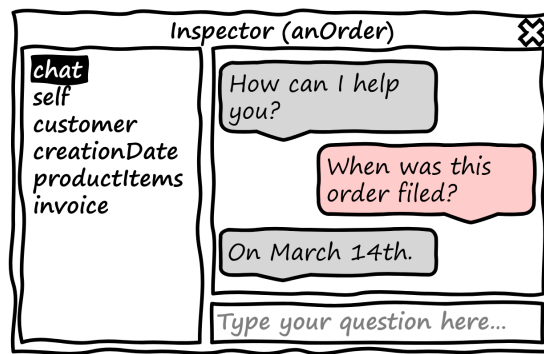


Figure 4.5.: Possible integration of a conversational semantic interface into a traditional object inspection tool. Programmers can ask conceptual questions about objects in natural languages besides inspecting their internal state.

domain. Additionally, they can ask follow-up questions in the context of a conversation without repeating or editing their original thoughts, like in a real-life conversation between two programmers.

Semantic messaging in scripts Scripting is another popular starting point for exploration. Especially programmers who are proficient in the programming language and familiar with the protocols of a system often prefer message sending through scripts to manually exploring object graphs through inspection tools. For example, the following Smalltalk script retrieves the product with the highest quantity from a list of order items:

```
(self orderItems detectMax: #quantity) product.
```

We propose an extension to object-oriented scripting languages that allows for *semantic messaging*: similar to pseudocode, programmers can write semantic messages with fictitious names (that do not even have to match the vocabulary of an object) to express their intents but send these messages to objects just like regular messages. Unlike regular messages, semantic messages do not require an implementation at the receiver object but get processed by an exploratory programming agent, which interprets the message as a question and internally inspects the object or sends it regular messages to determine a return value. For instance, the above example could be expressed as any of the following:

```
self mostOftenBoughtArticle.  
self orderItems theOneWithHighestAmount.
```

Like regular messages, semantic messages can also pass arguments, such as:

```
aProduct numberOfSalesTo: aCustomer.  
aProduct numberOfSalesFrom: 'Q3 2023' to: 'Q4 2023'.
```

4. *The Semantic Exploration Kernel*

Thus, programmers can maintain their scripting flow when asking questions even if they are unaware of certain protocols, and they do not have to express questions using specific protocols and syntax or implement algorithms for more complex problems.

*

Semantic object interfaces allow programmers to delegate parts of their research process to the programming system by expressing conceptual questions, letting an exploratory programming agent research them autonomously and map them to low-level experiments, and receiving conceptual answers in natural language.

Chapter Summary

We have presented the design and architecture of the *semantic exploration kernel*, which powers the semantic workspace by providing different functionalities based on semantic technologies. We have described the *suggestion engine*, which uses a *blackboard framework* for organizing suggestion artifacts and strategies, and the *exploratory programming agent*, which provides *semantic object interfaces* for asking semantic questions and autonomously reasons and experiments to answer these questions.

5. Suggesting Experiments with Semantic Retrieval

In this chapter, we describe how the suggestion engine creates different types of experiments to augment the exploratory research process of programmers. We provide an overview of the different types of artifacts and strategies in the suggestion engine. We explain how it implements several strategies for searching artifacts by using semantic retrieval, compare different ranking approaches for sorting and filtering suggestions, and describe how the suggestion engine incorporates the exploratory programming agent for suggesting code expressions.

5.1. Artifacts and Strategies

The suggestion engine defines several types and roles of artifacts and several strategies for recording and linking different kinds of information from the programming system.

We distinguish between three groups of artifacts:

INPUT ARTIFACTS are captured from the original exploratory session and experiments of the programmer.

This includes code artifacts such as *methods* and *classes* that the programmer is browsing or editing, *do-it scripts* that they are writing or executing, and other types of information such as *notes* and *bug reports* that they are interacting with.

RETRIEVED ARTIFACTS refer to any piece of information that is available in the programming system and has been identified by the suggestion engine as potentially relevant for the next steps of the programmer's research process.

Most retrieved artifacts relate to the implementation of systems: *users* of code artifacts such as message senders, class users, and variable references provide context about their tasks and interfaces. *Similar code artifacts* exemplify how other methods and classes solve related problems, use similar protocols, or implement similar interfaces. From the latter, *correlated code artifacts* can be extracted to summarize common building blocks of similar solutions, such as frequently sent messages, executed methods, instantiated

5. Suggesting Experiments with Semantic Retrieval

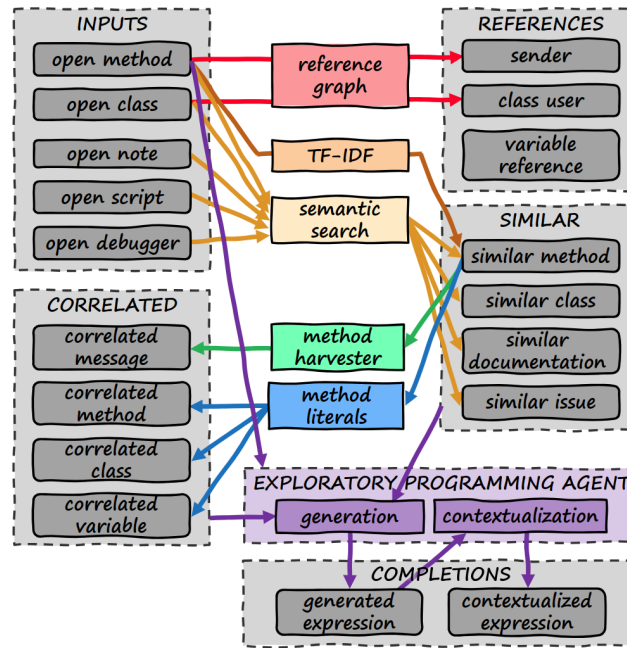


Figure 5.1.: Merge graph of all available artifact types and strategies in the suggestion engine. Gray boxes represent artifact types (grouped by their roles^a), colorful boxes represent strategies, and arrows display possible inputs and outputs. Arrows starting or ending in a group of artifacts or strategies refer to all of its items. For sake of simplicity, we do not distinguish optional and required inputs in this figure.

^aTechnically, each gray box represents a tuple of an artifact type and an artifact role. For example, “open method”, “sender”, and “similar method” are all artifacts of the type “method” in different roles.

classes, and referenced variables. A programmer who is editing a method has a higher probability to use some of these correlated artifacts.

Beyond code artifacts, the suggestion engine might also find *documentation artifacts*, *revision notes*, *bug reports*, or other forms of communication from information sources that are related to the development of the software system.

GENERATED ARTIFACTS are new experiments that the suggestion engine has synthesized.

This includes generated and contextualized *code expressions* based on similar and correlated artifacts. Generated code expressions can range from single message sends to entire class definitions and method implementations. However, the suggestion engine might also generate other types of artifacts to outline cluttered implementations, summarize long documen-

5. Suggesting Experiments with Semantic Retrieval

tation artifacts, or suggest a list of steps to debug an error based on prior developer communications.

For each suggestable role of artifact, the suggestion engine defines one or multiple strategies that process and combine prior artifacts and produce new artifacts. For example, we (i) implement multiple strategies for searching similar methods based on their intentions and interface usage; (ii) provide different ways to extract correlated messages from methods based on static and dynamic call graphs; and (iii) employ different LLMs and heuristics through the exploratory programming agent for combining retrieved artifacts into new code expressions. Figure 5.1 provides an overview of all available types of artifacts and strategies in our prototype.

5.2. Semantic Search

In the following, we describe multiple strategies for finding similar artifacts (such as similar methods) and correlated artifacts that are based on the former (such as messages that are frequently sent by similar methods).

Similarity Search

A central strategy for similarity search in the suggestion engine is *semantic search using document embeddings* [92, 97]. For this, we embed each query (i.e., each input artifact) using a text embedding model¹; search a vector database of all other available artifacts (such as methods in the system) for documents with nearby embeddings, and return the top k most similar results. Thus, we *reconstruct* the programmers' intentions by mapping their experimental artifacts into an embedding space, and we *anticipate* their next steps by finding similar artifacts in this space.

However, embedding-based search is only applicable to a limited extent: first, computing and storing embeddings involves noticeable costs in terms of time, storage, and financial expenses. Second, the embedding search might be too sensitive to the syntactical format of documents unless the model is specifically trained or documents are preprocessed, which can reduce the recall when comparing heterogeneous artifacts (such as the implementation and documentation of the same method). Similarly, when queried with an unfinished draft of a method or script, embedding-based search will primarily find artifacts of a similar length—such as stub methods—rather than fully implemented methods.

¹In our prototype, we use OpenAI's text-embedding-3-large model for embedding any artifacts and choose a reduced dimension size of 256 (see page 87).

5. Suggesting Experiments with Semantic Retrieval

To overcome these limitations, we provide another search strategy based on the TF-IDF (term frequency and inverse document frequency) metric [103]. This metric is based on the assumption that the less often a term occurs in a corpus of documents, the more important each occurrence of this term in a particular document is for the semantics of this document. For example, stopwords in natural language (e.g., common words such as “and” or “is”) or keywords and general-purpose identifiers from standard frameworks in programming languages (e.g., `self` or `ifTrue:` in Smalltalk) generally have a low significance for specific documents, while rare words or method names are more commonly associated with specific domain concepts.

Thus, we can estimate the similarity of two documents by comparing the occurrences of each term (or *term frequencies*) in both documents, weighted by their rareness (or *inverse document frequencies*). For this, the term frequency of a term t in a document d , the inverse document frequency of a document d in a corpus D , and the TF-IDF score are commonly defined as follows, with $d(t)$ indicating the number of occurrences of t in d :

$$\text{tf}(t, d) = \frac{d(t)}{\sum_{t' \in d} d(t')} \quad (5.1)$$

$$\text{idf}(t, D) = \log \frac{|D|}{|\{d \in D \mid d(t) > 0\}|} \quad (5.2)$$

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \cdot \text{idf}(t, D) \quad (5.3)$$

By combining the TF-IDF scores of a document for all terms in the corpus into a vector, we can embed the document into a high-dimensional embedding space. This allows us to compare two documents based on their cosine similarity analogously to other types of embeddings.

Other than transformer-based embeddings, TF-IDF vectors are sparse and ignore the mutual relation of words in a document as well as the semantics of words (such as synonyms and homonyms). However, they can be simpler to calculate, especially in the context of ranking similar documents to a reference document, where TF-IDF scores do not need to be calculated for every term.

Example. In a corpus with the three documents “Squeak is a system for programmers”, “Smalltalk is a programming language”, and “The squeak mouse is a birthday present for the dog”, a TF-IDF search will consider the first and the third document most similar to each other due to their shared use of the term “Squeak”. No pair of documents will be considered more similar due to their shared use of the terms “is” and “a”, because these terms appear in every document of the corpus. Since they use different inflection forms and hyponyms, the first two documents will not be considered similar.

5. Suggesting Experiments with Semantic Retrieval

In our prototype, we use TF-IDF search for methods based on the selectors and references to classes and variables in a method retrieved from its bytecode. We apply a binary weighting scheme for term frequencies (simplifying their definition to $\text{tf}(t, d) = 1_{\{d(t) > 0\}}$) and exponentially weight inverse document frequencies with a power of 4. In our experiments, this approach commonly yielded relevant results and showed a sufficient performance (typically 250 milliseconds for 25 MB of source code) even without maintaining an index.

Correlation Search for Code Artifacts

After finding similar artifacts to a set of input artifacts, we can extract frequent parts from them. In statistical terms, we say each of these parts *correlates* with the functional—or semantic—characteristics of artifacts like the input artifact. For code-related artifacts such as methods or documentation, their parts are the different code artifacts used by the similar artifacts, such as invoked methods, instantiated classes, or referenced variables. Their cumulative occurrence suggests a higher probability that programmers might also want to consider them for the input artifacts they are writing. This resembles the concept of *collaborative filtering* (“customers who liked this also liked...”), though this studies interactions of users with artifacts, while our approach analyzes dependencies between (different types of) artifacts [105].

To extract used artifacts from a given method, we employ two different data sources: a *static dependency graph* and a *dynamic dependency graph*.

The *static dependency graph* is modeled from the terms in a method, which are accessible from the abstract syntax tree (AST) or bytecode of the method. In our prototype for Smalltalk, we extract all literals from the bytecode of a method and filter them for message selectors, local variable references, and bindings to classes and global variables.

However, this approach only offers limited insight into the actual methods that are executed through message sends due to the dynamic dispatching of late binding programming languages (see [footnote 2.1 on page 12](#)). Using a *dynamic dependency graph*, we enrich this information with the execution context from Babylonian examples for each similar method artifact. In our prototype, we employ a *method harvester*, which runs all examples in an instrumented interpreter using SIMULATIONSTUDIO to construct this graph.² Thus, we can suggest a set of concrete methods that are executed by many similar methods.

We rank correlated artifacts based on their number of occurrences in the similar artifacts, weighted by the relevance of the latter.

²Inspired by the *type harvesting* approach of [19].

Example. A programmer is writing a script to create a red circle. Their incomplete draft looks like this:

```
circle := Circle new.  
color := Color red.
```

Based on the used names, the suggestion engine identifies similar methods such as:

- High similarity (3 common terms):

```
circle := Circle new.  
circle color: Color green.  
circle border: #thick.
```

- Moderate similarity (2 common terms):

```
triangle := Triangle new.  
triangle color: Color green.  
triangle shadow: true.
```

- Low similarity (1 common term):

```
rectangle := Rectangle newSquare.  
rectangle borderColor: Color blue.
```

From these similar methods, the suggestion engine suggests the following most relevant new correlated artifacts:

- #color: (used in 1 highly similar and 1 moderately similar method)
- #green (used in 1 highly similar and 1 moderately similar method)
- #border: (used in 1 highly similar method)
- #shadow: (used in 1 moderately similar method)

Thus, the programmer can complete their script by choose from the most likely suggestions.

5.3. Semantic Ranking

Many strategies produce a number of suggestions that is neither manageable for human programmers nor for exploratory programming agents using state-of-the-art LLMs. This is because semantic search methods, unlike traditional search methods, usually do not provide a binary classifier for the relevance of a document but compute a continuous score (e.g., the cosine distance in embedding-based search). Thus, we need to sort and filter suggestions before returning them to the requestor. We address both requirements through a *ranker* in the suggestion engine, which computes a ranking order

5. Suggesting Experiments with Semantic Retrieval

for all suggestions and returns the first k suggestions from this ranking to the requestor.

The requirements for ranking orders are various and depend on the context of the suggestions. Some roles of suggestion artifacts, such as similar or correlated artifacts, have a relevance score to a reference artifact, which should influence their ranking (e.g., to select the most similar artifacts). In other situations, such as when suggesting users of a class or method, suggestions should explicitly provide a broad overview of all available data and not respect any reference artifact. Between both extremes, gradations exist: for example, the exploratory programming agent might request senders of a method in the context of a broader topic, such as users of the `WriteStream»nextPut:` method in Squeak that are related to date-formatting tasks. In this situation, both suggestions relevant to the problem and suggestions exemplifying different uses of the protocol are desired.

We identify four *objectives* for ranking suggestions, which must be balanced differently depending on the context in which the suggestion engine is invoked. While we describe each objective with a formula, we have not used these formulas for a rigorous quantitative evaluation of different ranking methods but only provide them for mathematical intuition.

RELEVANCE A ranking must contain the most relevant suggestions from the corpus, such as those that are most similar to the reference artifact.

Formally, we operationalize this objective by requiring that for every prefix L of the ranking, a ranking method must maximize the sum of relevance scores R_i from all contained artifacts:

$$\text{Relevance}(L) = \sum_{i \in L} R_i \quad (5.4)$$

IRREDUNDANCY A ranking must not contain multiple highly similar suggestions that do not add value to the exploration, such as different almost identical versions of a method.

We measure this objective by counting the number of suggestion pairs (i, j) in the ranking that do not exceed a small threshold θ for their mutual similarity $S(i, j)$:

$$\text{Irredundancy}(L) = |\{i, j \in L \mid S(i, j) > \theta\}| \quad (5.5)$$

DIVERSITY A ranking must contain diverse suggestions that display highly different types and topics from the corpus.

We formalize the diversity of a ranking by summing up all mutual distances between two suggestions. Unlike irredundancy, which merely avoids

5. Suggesting Experiments with Semantic Retrieval

duplicates, this objective explicitly promotes outliers from different regions of the embedding space:

$$\text{Diversity}(L) = \sum_{i,j \in L} 1 - S(i, j) \quad (5.6)$$

REPRESENTATIVENESS A ranking must represent the full corpus to the best possible extent by containing suggestions that cover the entire distribution of the corpus.

While a comprehensive operationalization of this objective would provide little intuition, we can approach it by comparing statistical properties (such as the mean and standard deviation) of the ranking to that of the corpus and requiring the difference to be minimal. For a more elaborated measure, we could require a minimal Kantorovich distance [106], which indicates the cost of transforming the distribution of the corpus into that of the ranking.

To meet all requirements, we present four ranking methods and discuss how they balance all objectives differently (fig. 5.2): *top-k selection*, *probabilistic sampling*, *clustering*, and *probabilistic sampling from clusters*.

Top-k selection We sort all suggestions based on their relevance score and select the k most relevant artifacts. This approach maximizes the relevance of the ranking. However, it can also maximize the redundancy of results if the relevance score depends on the (transitive) similarity of artifacts. At the same time, diversity and representativeness are typically minimized because the selection explicitly focuses on most similar artifacts.

Probabilistic sampling We convert the relevance scores of the corpus into a probability distribution by scaling each relevance score exponentially to a probability p_i :

$$p_i = \frac{\exp \frac{R_i}{T}}{\sum_j \exp \frac{R_j}{T}} \quad (5.7)$$

Based on this probability distribution, we randomly sample k unique artifacts. Through a temperature parameter T , the probability distribution can be sharpened or flattened, modeling the ranking closer to top-k selection or uniform random selection, respectively. This approach corresponds to the *softmax* function used in several machine learning applications [107].

In comparison with all other discussed methods, probabilistic sampling maximizes the diversity of the ranking if a medium to high temperature is used. Similarly, the irredundancy or representativeness of results is slightly better than for top-k selection. However, results are significantly less relevant to the query.

5. Suggesting Experiments with Semantic Retrieval

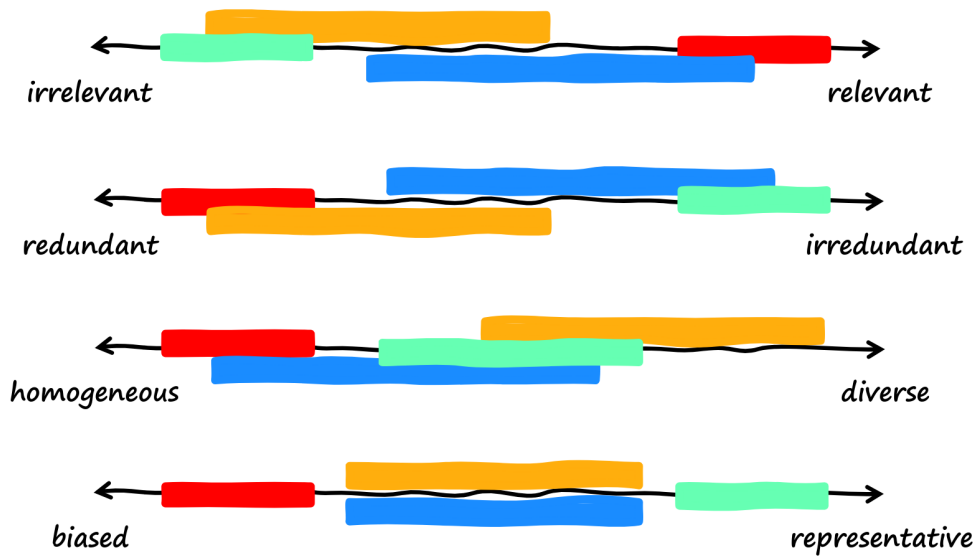


Figure 5.2.: The discussed methods for semantic ranking achieve different trade-offs between our defined objectives. Top-k selection (**red**) delivers the most relevant ranking but selects redundant, homogeneous, and non-representative items. Probabilistic sampling (**yellow**) maximizes diversity, while clustering (**green**) can improve the irredundancy and representativeness of items. With probabilistic sampling from clusters (**blue**), we find an acceptable balance of all objectives.

Note that this sketch shows a broad comparison of methods only, and exact metrics would heavily depend on the distribution of original suggestions and on the choice of hyperparameters (such as temperature) for each method.

Clustering We cluster the corpus into k groups by using a partition-based algorithm such as k-means. From each cluster, we select the most representative artifact (i.e., the center of the cluster).

Cluster-based selection maximizes the irredundancy and improves the representativeness of the ranking. However, it does not honor the relevance of artifacts. Also, cluster centers present a lower diversity than probabilistic sampling as they neglect outliers.

Probabilistic sampling from clusters We group the corpus into k clusters, assign a probability to each cluster based on the relevance score of its center, and sample random clusters. For each occurrence of a cluster, we select the next most relevant artifact from it. Similar to plain probabilistic sampling, this method can be controlled through a temperature parameter.

Cluster-based probabilistic sampling provides a trade-off between the other discussed methods with respect to all considered objectives: in our experi-

5. Suggesting Experiments with Semantic Retrieval

ments, it yielded a moderate relevance of suggestions that is between that of top-k selection and plain probabilistic sampling, a moderate diversity close to clustering, and a sufficient irredundancy and representativeness.

Failed attempts. We also experimented with two other ranking methods that did not yield usable results for us:

Explicitly amplifying diversity We computed a diversity score for each artifact as its average distance to all other artifacts and added that to the artifact score. Despite experimenting with different weights and scalings, we were not able to create rankings that would combine relevance and diversity in a useful way.

Clustering from an RBF-transformed embedding space We transformed the embedding space of the corpus using a radial basis function (RBF) centered around the reference artifact before clustering it. We could not confirm our hypothesis that this approach would balance relevance and representativeness. In our experiments, the complexity and sensitivity of parameters rendered this approach impractical.

In our prototype, we found probabilistic sampling from clusters to balance the different objectives in the most useful way and to be versatile for most applications. For some use cases where no reference artifact has been specified, we use the plain clustering method instead.

5.4. Semantic Generation

Another group of strategies in the suggestion engine implements the generation of semantic completions, that is, synthesized artifacts that continue a programmer's draft of a code artifact, documentation artifact, or a similar artifact. These synthesized artifacts are based on other types of artifacts such as similar methods, correlated classes, or documentation excerpts. The actual generation is delegated to the exploratory programming agent, which employs generative LLMs for creating semantically relevant completions (see [chapter 6](#)).

However, the requirement for interactive completions conflicts with the high latencies and monetary costs of the exploratory programming agent: on the one hand, semantic completions should not introduce temporal distances in the research process of programmers but be updated with a high frequency to always reflect the latest input from the programmer. On the other hand, the agent fundamentally depends on invocations of resource-intensive generative LLMs that typically involve latencies of several seconds for complex requests.

To solve this conflict, we divide the generation of semantic completions into two stages. In the first stage, the agent performs a full generation of

5. Suggesting Experiments with Semantic Retrieval

completions based on extensive context from other suggested artifacts and a comprehensive chain of thought. In the second stage, the prior generations are *recontextualized* to adjust them to minor changes in the original draft. For example, recontextualization would remove prefixes from the completions that the programmer has already typed manually, apply recent renamings in the draft to completions, or align them with other recent restructurings or updates by the programmer. This second stage requires less context (only the prior completion and the new draft) and reduced reasoning. This allows the agent to minimize the request to the LLM and employ a smaller model. Additionally, recontextualization can be implemented (partially) through non-semantic heuristics such as substring matching and AST merging.

Through this separation, the suggestion engine executes the first generation stage with a low frequency only when the programmer performs larger changes to a draft (such as adding several new lines) or manually requests new generations, while the second stage can be executed with a high frequency but lower latencies and costs.

Chapter Summary

The suggestion engine anticipates experiments of programmers by capturing the context and actions of programmers as *artifacts* and defining *strategies* that create further artifacts based on the former. We have detailed different strategies to search the system for similar artifacts based on document embeddings and the TF-IDF metric as well as for *correlated* artifacts based on frequent parts of similar artifacts. We have discussed different objectives and methods for *ranking* artifacts. To optimize the suggestion of semantic completions, we have proposed a *two-stage generation approach* based on the change rate of programmers.

6. Building an Exploratory Programming Agent with GPT-4o

In this chapter, we describe our implementation of semantic object interfaces through a conversational autonomous agent. In our prototype, we use OpenAI’s LLM GPT-4o,¹ which ranks among the state-of-the-art models for our required capabilities such as problem solving and code writing at the time of writing [99].

We implement the agent’s policies through prompt engineering and map the system interface to a set of functions that the LLM can call.² Finally, we optimize the agent’s performance for generating semantic code completions.

6.1. Implementing Policies through Prompts

We define a set of *policies*, which define the behavior and the communication styles of the exploratory agent:

IDENTITY AND CONTEXT: The agent shall identify as an exploratory programming agent who supports the programmer in exploring an object in a system. In conversation mode, the *assistant* shall also identify as the reference object itself, allowing users to address the object in the second person within the chat and thus increasing the experienced semantic immediacy³

TRAITS FOR PROBLEM SOLVING AND TOOL USAGE: The agent shall adopt a structured problem-solving approach by explicating ideas and steps in advance. It shall check the results of experiments and iterate as necessary before providing a final answer to the programmer. When referring to object protocols, the agent shall automatically browse the implementation or senders of messages whose behavior is unclear, and it shall automatically test message sends by executing them before suggesting them to the programmer.

OUTPUT FORMAT: The agent shall provide brief and concise answers and avoid long sentences unless requested otherwise. When addressed through

¹Used in the version gpt-4o-2024-05-13.

²We provide a full conversation including all system prompts and function calls for the example from fig. 8.5 in appendix B.

³This follows the conversation style of message sends which are usually named in imperative or interrogative speech (such as the cascade-terminating message `#yourself` in Smalltalk).

6. Building an Exploratory Programming Agent with GPT-4o

semantic messages, the agent shall answer an object instead of natural-language text. Objects can be fetched from the system or be created by the agent; in particular, primitive values (such as numbers and booleans) or dynamically structured JSON objects are preferred.

To configure the agent to follow these policies, we describe them comprehensively in the system prompt of the agent conversation (fig. 6.1). While we iterated our prompt design several times, we did not conduct a systematic, evaluation-based approach to prompt engineering [113]; yet our prototypical agent already showed promising results.

Bootstrapping the exploration We initialize the internal conversation of the agent with a sequence of pre-generated messages that demonstrate the intended behavior of the agent. These pre-generated messages feature inner monologue and (resolved) function calls by the assistant. By this, they illustrate the agent’s steps for getting a first overview of the object. For example, our default pre-generated conversation prefix shows how the agent retrieves a textual representation of the provided object and enumerates its instance variables (see the “bootstrapping the exploration” region in fig. 6.1).

Additionally, we inject *hardcoded semantic context* about the object into the pre-generated conversation prefix through system messages in natural language. This context includes information about the role or particular messages of an object. For example, for the class `Context`, which represents a stack frame in a debugger (see fig. 8.8), we provide a brief situational explanation and point the agent to relevant protocols for querying the entire debugger stack.

This pre-generated conversation prefix serves several purposes: first, it provides some broad context about the object to the agent within the first invocation, which serves as the base for further experiments by the agent or maybe contains the necessary information already. Second, the first likely actions of the agent are anticipated, improving the average response time of the agent. Third, the prefix “stimulates” the agent toward an exploratory mindset, engagement in inner monologue, and eagerness to perform several experiments. Thus, pre-generated conversation prefixes present a hybrid of zero-shot chain-of-thought prompting [90], few-shot prompting [77], and retrieval-augmented generation [92].

6.2. Designing System Interfaces for Automatic Agent Experiments

To define a set of functions through which the agent can access the system, we imitate the actions that programmers can take in traditional exploratory

6. Building an Exploratory Programming Agent with GPT-4o

HEADER	
Exploratory programming agent	
System: You are an exploratory programming agent...	• <i>identity</i>
System: You can call the following functions...	• <i>interface description</i>
System: To solve a task, you should...	• <i>rules and traits for problem solving</i>
Conversation mode (optional)	
System: You are an object...	• <i>object identity</i>
System: Keep your answers brief...	• <i>output format</i>
Semantic messaging (optional)	
System: You must call the evalAndReturn function...	• <i>output format</i>
System: Format the return value as...	
Bootstrapping the exploration	
System: This object represents...	• <i>hardcoded semantic context</i>
Assistant: To understand this object, I will first...	• <i>zero-shot chain-of-thought</i>
Assistant: eval("self printString")	
Result: an Object(12345)	• <i>initial object context</i>
Assistant: eval("self allInstVarNames")	
Result: #('foo' 'bar')	
BODY	
User: What does this object...?	• <i>user question</i>

Figure 6.1.: Schematic prompt design for conversations of the exploratory programming agent with the user and the system. We translate policies for the identity, strategies, and output formats of the agent into detailed instructions. Through pre-generated assistant messages and tool calls, we stimulate the agent to engage in inner monologue and several experiments.

programming systems. For example, we allow the agent to *inspect* the state of an object similarly to an inspector tool by requesting variable values; *browse* the source code of the system similarly to a code browser by requesting its packages, classes, and methods; or *send messages* to objects by evaluating scripts. Table 6.2 shows the complete function interface of our prototypical agent for Squeak/Smalltalk. Below, we discuss two challenges in designing these functions: the granularity of function calls and the programming language proficiency of the agent for evaluating code.

Granularity of function calls An important trade-off regards the amount of information that is requested through a single function call: for example, when browsing a class, the agent could either first request a list of protocols in a class and then request the message names within relevant protocols, or request the entire list of message names for all protocols at once. In our prototype, we generally choose a medium-to-coarse-grained function design because for many cloud-based LLMs such as OpenAI's GPT models, the cost of processed tokens is quadratic in the number of sequentially requested function calls

6. Building an Exploratory Programming Agent with GPT-4o

Table 6.2.: The function interface that connects our exploratory programming agent to the Squeak/Smalltalk system. Most browsing functions are designed to return extensive information, reducing the need for subsequent function calls.

Function	Description
<code>eval(expression)</code> <i>Example:</i> <code>eval("self customer")</code>	Evaluate a Smalltalk expression in the context of the explored object and return the result or error. Can be executed in isolation.
<code>evalAndReturn(expression)</code>	Evaluate a Smalltalk expression in the context of the explored object and pass back the result to the sender of the original semantic message. Only available if the agent was invoked through a semantic message.
<code>browsePackage(packageName)</code>	Return a hierarchical list of classes within a package.
<code>browseClass(className)</code>	Enumerate all methods defined on a class or one of its superclasses or their metaclasses, grouped by the defining class and the method category (protocol) within the class organization.
<code>browseMethod(className, selector)</code>	Retrieve the source code of a method defined in a class.
<code>browseSenders(selector[, query])</code> <i>Examples:</i> <code>browseSender("printOn:")</code> <code>browseSender("printOn:",</code> <code>↳ "date yy-mm-dd")</code>	Search the system for all methods that send messages with the name of a selector and return a representative sample of these methods. If an optional query argument has been provided, rank the methods based on their relevance to the query (see section 5.3).

(because every newly requested function call requires a new API request and processing of the prior conversation).

Evaluating code Unspecialized LLMs are not always proficient in particular programming languages: for example, when writing Smalltalk code, GPT-4o often produces syntax errors and shows insufficient knowledge of standard libraries such as the `Collections` package. To support the model in correcting its own errors, we extend a small number of built-in error messages of a system with practical suggestions based on typical faults of the model [73]: for example, because GPT-4o often forgets necessary brackets when chaining message sends, we extend all `MessageNotUnderstood` errors from the system with an explaining comment and provide an example of correcting an incorrect message chain, which the model then considers in the next attempt.

We reduce the dependency of the agent on language proficiency by exposing most interfaces through dedicated functions instead of instructing the model to access them through evaluating reflection code. For example, while it could be elegant to have the agent retrieve the protocols of a class by calling

`eval("(Smalltalk at: aClassName) organization categories");`⁴ we offer a dedicated `browseClass()` function for this purpose instead.

To avoid dangerous side effects of AI-generated code such as data loss and system crashes, it is also possible to evaluate all requested scripts in a lightweight sandbox or only apply their side effects after manual review. However, this imposes the additional complexity of managing “different realities” on the agent [44], and the isolation layer can impact the overall performance of experiments. In our prototype, we provided an option to disable the sandbox and left it disabled most of the time, as we only observed a small number of unintended side effects (see page 83).

6.3. Optimizing for Semantic Completions

As described in section 5.4, resource consumption and pecuniary cost are particularly critical for generating semantic completions. Because up-to-date and immediate completions are desired throughout the entire programming session, this even holds true when following the two-stage generation approach described there.

To optimize the exploratory programming agent for this use case, we provide the LLM with an extensive pre-generated context but disable its function interfaces and limit the research process of the agent to a single invocation of the model. That is, we eliminate the autonomy of the agent to improve its performance.

In our prototype for semantic code completions, we statically (i.e., at development time) anticipate a range of likely experiments required by the agent and dynamically (i.e., prior to the only invocation of the LLM) execute them and present their results to the agent for stage-1 generations. These pre-generated experiments include browsing the definition, protocols, and messages of the receiver class of the current draft, correlated classes provided with the same information, and the source code of correlated methods. If a current receiver object is available (e.g., when the programmer is typing their draft into the script pane of an inspector), we also include its full state (i.e., all instance variables and truncated variable fields).

Similarly, we attempt to provide dynamic context for all displayed classes by including previews for their message results through a form of speculative execution (i.e., we locate sample instances of each class in the image and send each unary message to them in a sandbox). To improve the ability of the agent to follow the given task, we provide it with a one-shot prompt of processing a similar request. To mitigate the limited proficiency of GPT-4o regarding the

⁴At the time of writing, a similar technique is used in ChatGPT when browsing long documents.

6. Building an Exploratory Programming Agent with GPT-4o

Code completion agent	
System: You are a code completion agent...	• <i>identity</i>
System: You will complete a method...	• <i>task and rules</i>
System: Use the following information...	• <i>data description</i>
Example	
User: DateAndTime»yyymmddString ^	• <i>draft</i>
User: self: 2024-06-22T00:30:37.216061+02:00 class: DateAndTime utcMicroseconds: 1719009070988843 ...	• <i>receiver object state</i>
User: Magnitude subclass: #DateAndTime... ('accessing' getSeconds -> 1870 setSeconds:...	• <i>receiver class: definition and protocols with preview results</i>
User: Related classes and methods: ArrayedCollection subclass: #String... example: 'hi' ('accessing' byteAt: byteSize -> 2 ...	• <i>correlated classes: definition and protocols with preview results</i>
User: SequenceableCollection»streamContents: blockWithArg ^ self new...	• <i>correlated methods: definition and implementation</i>
Text class»exampleWithNumber: x <primitive: "primitiveCopyBits"...	
Assistant: DateAndTime»yyymmddString ^ String streamContents: [:stream self...	• <i>completion</i>
Task	
User: <information about task in the same format as above>	
System: Now complete this:	• <i>repetition of the task context</i>
User: <draft again>	

Figure 6.3.: Optimized prompt design of the exploratory programming agent for generating stage-1 code completions. Instead of enabling the agent to perform different experiments autonomously, we provide it with extensive pre-generated information and request a final answer in a single invocation of the LLM. (Fields with ellipsis (...)) have been abbreviated for this figure but appear untruncated in the LLM context.)

Smalltalk programming language, this shot also includes an extensive example of the Smalltalk syntax.

As for other invocations of the agent, we instruct it to engage in inner monologue before returning a single code completion. We request the LLM to return $k \in [5, 20]$ completions in parallel, which multiplies the cost for the outputs but not for the provided context, while latencies are typically not increased due to elastic scaling of the API service. Figure 6.3 displays the full prompt schema of our agent for generating a semantic code completion.

For stage-2 generations (recontextualizations), we invoke the smaller, more efficient GPT-4o mini model⁵ once for each previously generated stage-1

⁵Used in version gpt-4o-mini-2024-07-18.

6. Building an Exploratory Programming Agent with GPT-4o

Task	
System: Use the following code snippet...	• <i>task and rules (brief)</i>
User: <stage-1 expression>	• <i>previously generated code completion</i>
System: Now complete this:	
User: <updated draft>	• <i>task context</i>

Figure 6.4.: Optimized prompt design of the exploratory programming agent for generating stage-2 code completions. We use a smaller LLM to recontextualize the previous stage-1 expression based on the updated draft of the programmer.

completion, provide it with the previous completion expression as well as the updated draft of the programmer, and instruct it to adjust the previous expression (fig. 6.4).

Chapter Summary

We have implemented an exploratory programming agent for semantic object interfaces by *prompting* a GPT-4o model with a set of behavioral policies and initial research steps, connecting it to the system through *functions* for executing experiments, and pre-generating statically structured prompts to efficiently generate code completions.

7. Integrating the Semantic Workspace into Squeak

In this chapter, we describe how we integrate our prototypical implementation of the semantic workspace with the required semantic technologies and the Squeak/Smalltalk exploratory programming system.

To make semantic technologies accessible in Squeak, we use our SEMANTICTEXT framework, which provides a domain model for semantic retrieval and text generation, integrates different language models from the OpenAI API, and offers fundamental tooling for prototyping semantic applications. We provide a detailed presentation of the framework in [appendix A](#).

In the following, we briefly address particular design and implementation considerations regarding the feasibility and usability of each semantic tool in the semantic workspace: semantic suggestions, semantic completions, and semantic conversations.

7.1. Semantic Suggestions

We implement semantic suggestions through a *suggestion space* window that is placed as a vertical docking bar at the edge of the screen ([fig. 7.1](#)). To track the experiments of programmers, we observe their mouse and keyboard focus by using Morphic’s event filter mechanism. From the focused windows, we extract the currently browsed classes and methods or drafted scripts and notes as input artifacts for the suggestion engine.

In the suggestion space, we display similar code and documentation artifacts, correlated code artifacts, and an optional summary of them. Suggestions are updated continuously in the background as the programmer conducts experiments by browsing methods or writing code or notes.

To enable semantic retrieval of all classes and methods in the image, we maintain either of them in a semantic corpus (i.e., a vector store of SEMANTICTEXT).¹ We subscribe to the `SystemChangeNotifier` interface to incorporate updates from the system. We compute embeddings for new doc-

¹We truncate methods to their first 10 000 characters to exclude “data methods” that define constants such as multimedia data, since they might exceed the context window of the embedding model and rarely contain interesting, human-readable information. While a typical Smalltalk image still contains many shorter data methods, this heuristic already

7. Integrating the Semantic Workspace into Squeak

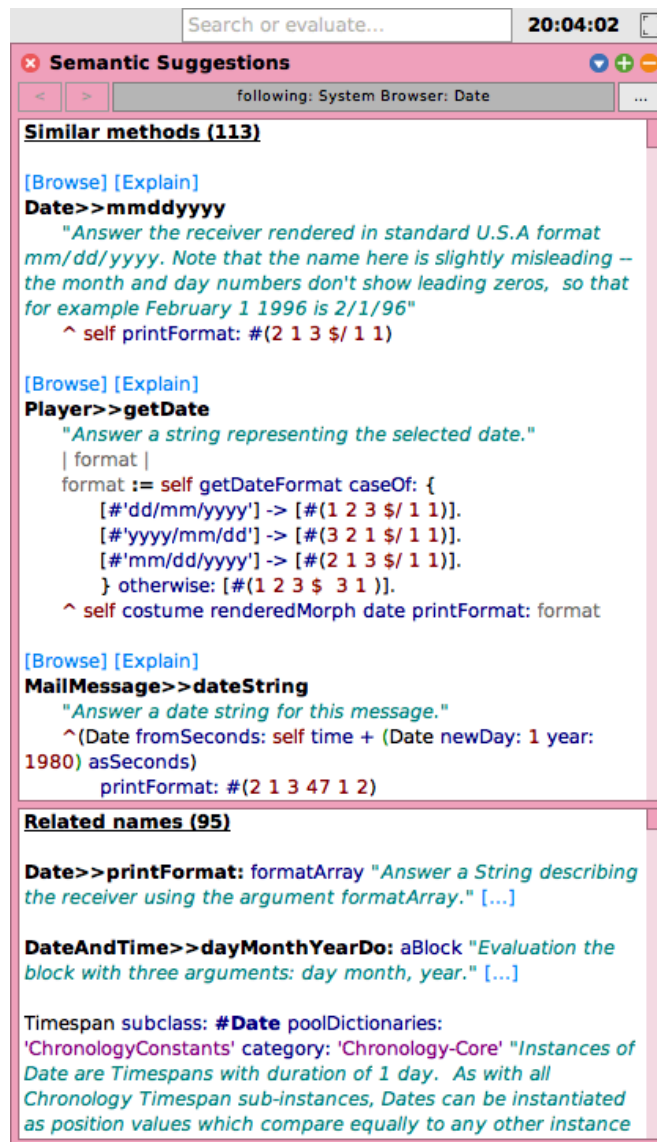


Figure 7.1.: The user interface of our *suggestion space* for displaying semantic suggestions in Squeak. It is docked to the side of the screen, tracks browsing and editing activities of programmers, and automatically displays suggested artifacts in separate panes. Programmers can derive inspiration from suggestions for their own applications or use the “browse” buttons to explore individual artifacts. Through the “...” menu, they can customize used strategies and displayed types of artifacts in the suggestion space.

uments asynchronously in the background after the current world cycle has ended (i.e., [aCorpus computeEmbeddings] future forkAt: Processor systemBackgroundPriority). This strategy allows for bulk-updating the corpus and avoiding the overhead of multiple requests to the OpenAI API after many code changes have been applied as part of a single operation (such as installing a package). At the same time, corpus updates are run in the background without noticeable lags.

7.2. Semantic Completions

Our implementation of semantic completions reuses the framework of the suggestion space for tracking the experiments of programmers.

For the user interface of completions, we use the `AUTOCOMPLETION` package² and employ its `ECEntryHook` interface (fig. 7.2). We contribute two new types of completion entries: *correlated names* and *generated expressions*.

First, we provide regular selectors and variable names based on correlated suggestion artifacts from the suggestion space and merge them with traditional `AUTOCOMPLETION` entries. This considers the original ranking of entries, which is only based on alphabetical order and most recently used date, and enhances it with their semantic relevance. We hook into the presentation of

²reduced the token consumption of the embedding model by approximated 13% in our experiments.

²<https://github.com/LeonMatthes/Autocompletion>

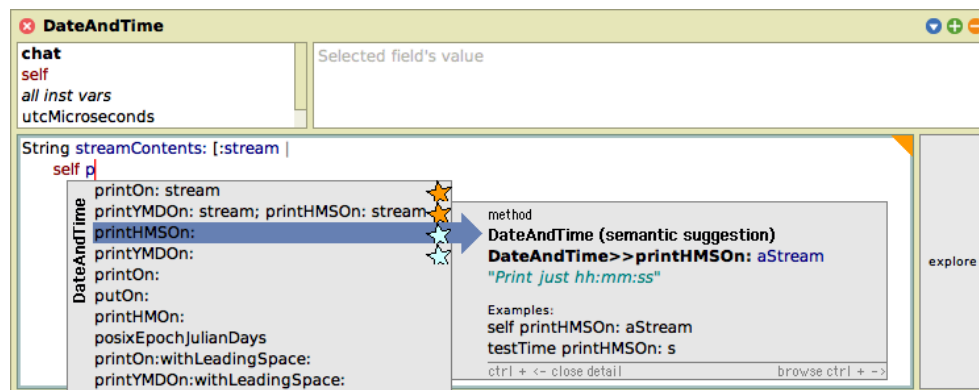


Figure 7.2.: Our integration of semantic completions into the `AUTOCOMPLETION` tool for Squeak, here invoked from an inspector on a `DateAndTime` object. Correlated names are displayed with a blue star and generated expressions are displayed with an orange star. Programmers can select an entry to read details such as documentation, usage examples, and a preview of its result, and press Tab to insert it into their editor.

entries to enrich suggested names with usage information mined from their preceding similar code artifacts as well as related documentation.

Second, we include contextualized generated expressions (see section 5.4) into the completion menu. Analogously to the suggestion space, semantic completions are computed asynchronously in the background to avoid noticeable delays in the user interface.

To generate a preview of the result of generated expressions, we run them in the editor context inside an isolated sandbox of SIMULATIONSTUDIO and include the results in the presentation of each entry.

7.3. Semantic Conversations

To implement the exploratory programming agent in our prototype, we define several specializations of the `SemanticAgent` class from `SEMANTICTEXT` (see section A.1). Each version of the agent contains instructions for a different type of semantic object interface as discussed in section 4.3: a conversation mode for object inspectors and a language extension for semantic messaging.

To add a new conversation mode to Squeak’s inspector tools, we redirect all requests to the `Inspector` class by the toolset to a decorator. This decorator inserts a new chat item at the beginning of the inspector’s field list, which references a conversation with our exploratory programming agent. Second, the decorator embeds a minimal version of the conversation editor GUI (graphical user interface) from `SEMANTICTEXT` when this field is selected (fig. 7.3).

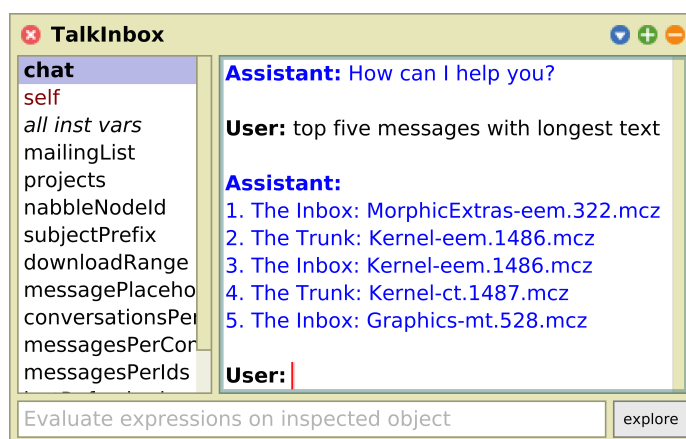


Figure 7.3.: Our integration of semantic conversations into Squeak’s inspector (here: “chat” item in the field list on the left). In this example, the user chats with an archive of the Squeak mailing list^a to identify exceptionally large posts.

^a<https://github.com/hpi-swa-lab/squeak-inbox-talk>

7. Integrating the Semantic Workspace into Squeak

To implement semantic messages in Squeak, we use the dynamic message-dispatch mechanism of Smalltalk's meta-object protocol by patching the method `Object>>#doesNotUnderstand:` [22, sec. 5.11] and forwarding all unknown messages to the agent. Through an additional policy and a change in the function design (that is, by requiring an eventual call of the `evalAndReturn()` function from table 6.2), we configure the agent to return Smalltalk objects instead of natural-language text. Because we desired a distinction between regular and semantic messages during our experiments, we also implemented two alternative forms for expressing semantic messages:

`SEMANTIC PROXIES` are constructed explicitly and override `#doesNotUnderstand:` to handle semantic messages:

```
self semanticProxy mostOftenBoughtArticle.  
aProduct semanticProxy numberOfSalesTo: aCustomer.
```

`THE ? AND ! OPERATORS` take the semantic message as an argument selector:

```
self ? #mostOftenBoughtArticle.  
pendingOrders ! #cancelItemsFromSpringSeries.
```

Here, the `?` operator indicates a declarative query for information. In contrast, the `!` operator permits side effects, allowing programmers to modify the state of objects in a semantic style.

While the `?` and `!` operators do not support additional arguments by design, we usually preferred them to semantic proxies in our experiments due to their reduced typing effort.

Chapter Summary

We have implemented the three tools of the semantic workspace in Squeak by using our `SEMANTICTEXT` framework for accessing semantic technologies. We have described several design decisions and implementation details for integrating suggestions, completions, and conversations into the programming interface of Squeak.

8. Applications

In this chapter, we illustrate how programmers can benefit from the different tools of the semantic workspace for augmenting typical exploratory activities. To this end, we describe two case studies: in the first one, a programmer uses semantic suggestions and semantic completions to prototype a GUI application. In the second case study, a programmer conducts a semantic conversation with a text object to explore its state, capabilities, and implementation details. Finally, we describe how semantic conversations can be integrated more deeply into traditional exploratory programming systems by leveraging the object-oriented interfaces of existing programming tools.

8.1. Prototyping User Interfaces with Semantic Suggestions and Completions

In this case study, we describe how a programmer builds a prototype for a simple counter application in the Morphic framework of Squeak with the help of semantic suggestions and completions.

To start the exploration, the programmer opens a workspace, types in a simple do-it for creating and showing a simple morph (i.e., a graphical object), and executes it:

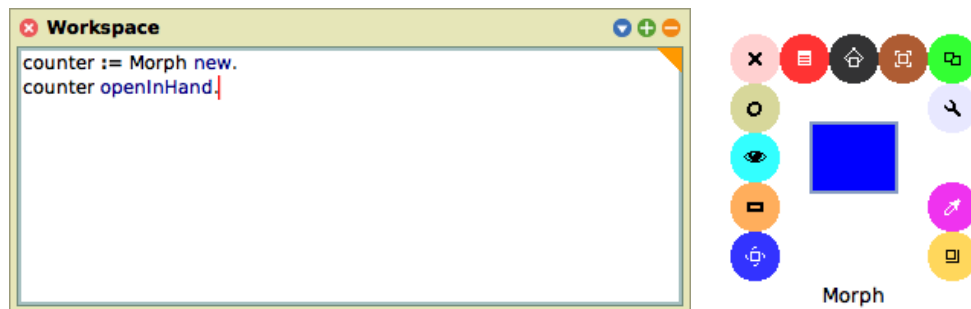


Figure 8.1.: Bootstrapping the prototyping of a Morphic counter application in a workspace.

As the programmer is viewing the plain default morph and the workspace is ready for further experiments, the programmer wonders about the different possibilities for visual customization that the morph offers. Traditionally,

8. Applications

answering this question would require them to browse the available protocols of the morph, check out the documentation of the class, or research different samples of similar existing Morphic applications.

However, once the programmer has entered the do-it, the *suggestion space* has already conducted plenty of such research in the background and now displays a summary of information about the morph's interface: in a list, the programmer can view different common selectors for modifying the appearance, layout, or composition of the morph. For each selector, the suggestion space displays the parameter signature of the message, any available documentation from the Morph class, and a short list of example usage snippets from other packages in the system:



Figure 8.2.: Using the suggestion space to browse relevant protocols of the class Morph for prototyping a user interface (shortened for figure).

8. Applications

Within this list, the programmer discovers the `#color:` selector and decides to adjust the color of their counter morph. To achieve that, they simply reuse one of the included usage examples and apply it in the workspace (`counter color: Color gray`). Analogously, they change the size of the morph by adding an `#extent:` message send. Thus, the programmer was able to answer their question about relevant protocols of the morph without searching the interfaces of the Morph class manually, seek inspiration from automatically collected examples, and understand how to use these protocols through the context of the provided documentation and usage snippets.

Next, the programmer wishes to add a border to the morph. From the examples in the suggestion space, they understand that borders can be configured through the `#borderStyle:` selector, but they do not yet comprehend all the different options that are available through the argument of this selector. Nevertheless, they begin by already typing an incomplete message send with that selector into their workspace. Traditionally, they now would be required to browse further examples or documentation related to this selector to learn more about possible arguments that can be passed.

However, the *semantic autocompletion* has detected the intent of the programmer through their typed prefix and automatically suggests different possible border styles in the completion menu of the text interface. Each border style is given through a different argument expression, exhibits different visual features such as width, dashes, and skeuomorphisms, and is displayed with a graphical preview of applying this style to the counter morph:

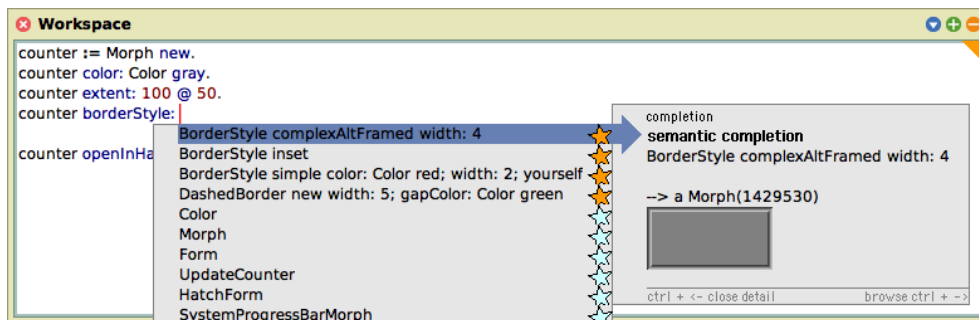


Figure 8.3.: Using the semantic autocompletion to explore different customization possibilities for borders of our Morphic UI prototype.

The programmer can browse these completion suggestions to examine the generated code, related documentation from the used classes and messages, and their effect, and accept one of the completions to insert it into the workspace. Thus, the programmer could explore and compare different possible designs based on the available interfaces while avoiding to manually browse existing usage samples, combine them, or transfer them to the context of their own script.

8. Applications

In a similar manner, the programmer can also proceed to add a label and a button to the counter morph. As the general overview in the suggestion space currently does not show any suggestions to these specific widgets, they start by explicating their intent through a comment in the workspace ("add label and button"). In response, the suggestion space and the semantic autocompletion provide them with suggestions for different types and layouts of labels and buttons for the counter morph.

After the programmer has chosen and applied any of these suggestions, the final step is to make their prototype functional by adding behavior to the button. To this end, they start by defining a new counter variable. Automatically, the semantic autocompletion scans the available protocols of the used widgets and suggests different code snippets that will trigger an increment of the counter variable and an update in the label for every click on the button. Again, the programmer compares the different implementations, chooses their preferred option, and executes it in the workspace to complete their first prototype of the counter application:

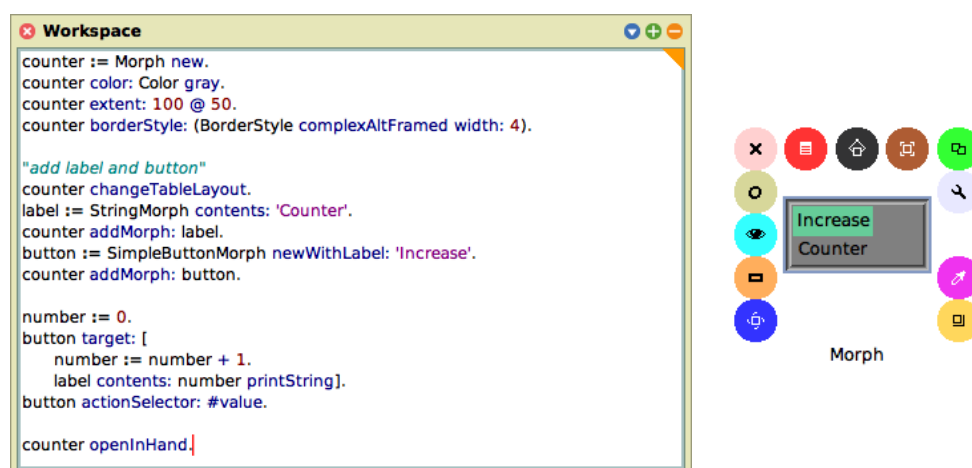


Figure 8.4.: The complete prototype of our counter application in Squeak.

In summary, the programmer could build a UI prototype in a more fluent process, spend less time, and achieve a better-grounded result by using semantic suggestions and semantic completions. They could remain in their flow of conceptualizing a UI design without being disrupted by having to manually research extensive interfaces and documentation or read and comprehend many incoherent and redundant examples. They could improve the quality of their prototype by considering suggestions based on a larger number of sources and comparing multiple distilled, distinct options with regard to their inner (code) quality and their visible effect.

8.2. Exploring Text Formatting through Semantic Conversations

In this case study, we describe how a programmer uses a semantic conversation to explore Squeak’s programming interface for creating formatted texts. In Squeak, the `Text` class represents a formatted text that is modeled through a string and a run-length encoded array of nested sets of instances of a `TextAttribute` class hierarchy that provide different instructions to the text renderer.

To start their exploration, the programmer discovers an existing `Text` object in the system that looks like this:

ABCDE1

To understand its design and behavior related to formatting, the programmer invokes an inspector on this text (fig. 8.5). First, they wonder what attributes are contained in the text. As the internal structure of nested collections looks slightly overwhelming, they switch to the *semantic conversation mode* of the inspector instead. Here, they enter the following question:

“What attributes are in this text?”

In response, the exploratory programming agent automatically inspects the internal structure of the text, iterates over the nested collections, and finally lists both present attributes correctly in the chat: a “`TextEmphasis` with code 1” and a “`TextEmphasis` with code 2”. This points our programmer to the `TextEmphasis` class but also motivates them to learn more about its features and representation. Thus, they type a follow-up question into the chat:

“What do these codes mean?”

Note that they can ask this within the context of the conversation—without needing to respecify the actual codes they are referring to or the class that defines them. The agent processes this question by automatically browsing the documentation of the `TextEmphasis` class, locating the relevant information in its class comment, and printing the correct list of all emphasis codes into the chat.

Finally, the programmer wonders how they can add other emphases to the text, so they ask for several code examples to italicize the entire object. In response, the agent automatically browses the protocols of the `Text` class,

¹Alternative description for accessibility: The first three letters are emphasized in bold, and the last letter is underlined.

8. Applications

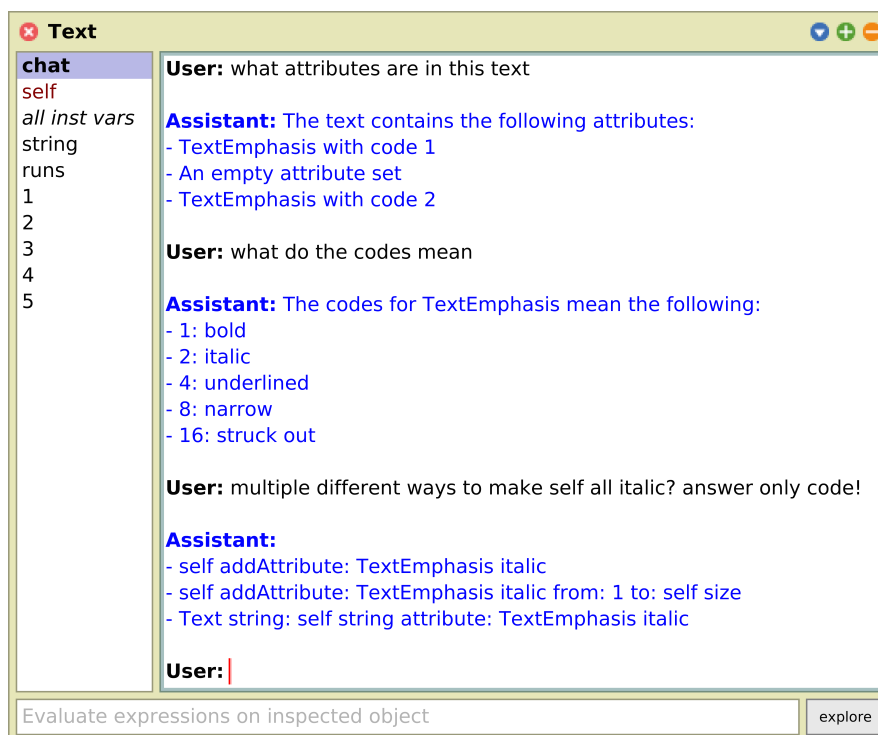


Figure 8.5.: Using the semantic conversation mode in an inspector to chat with a formatted Text object. Through the chat, the programmer can understand the attributes used for formatting and explore the available protocols for applying other formats to texts.

identifies and tests several possible messages, and provides three valid snippets to the programmer that would achieve the desired behavior. The programmer can work with these snippets, adjust them with the help of the agent or by themselves, or integrate them into their own program.

This case study shows that the conversation mode in inspection tools can be used to answer a wide range of questions: through functional questions, programmers can access, search, or summarize domain information. Another example of this is filtering items in a task management system based on their content (fig. 8.6). Through epistemic questions, programmers can get familiar with domains, explore systems and interfaces, and prototype ideas as working applications. In other settings, this could be used to study the different sorting protocols of collections, brainstorm and compare different options for formatting dates, or iteratively create visualizations.

In summary, programmers can use semantic conversations to ask semantic questions about objects, which the agent attempts to answer by extracting, analyzing, and synthesizing information. Thus, programmers can maintain their flow of exploring a domain from a conceptual perspective for a longer

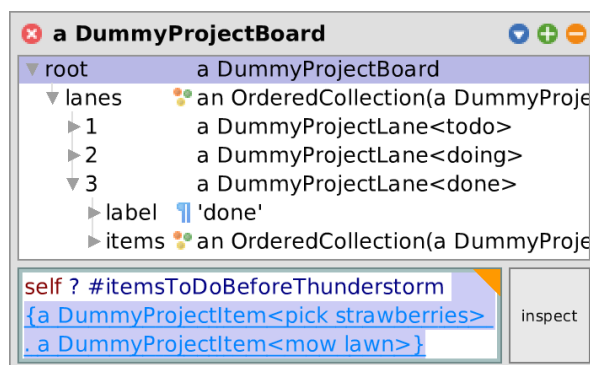


Figure 8.6.: Sending a semantic message to search and filter a small project management system for common-sense questions. The exploratory programming agent automatically explores the available protocols of the project management systems to retrieve items from the lanes of the project board and then filters them.

period of time without being overwhelmed or distracted by performing manual technical experiments.

8.3. Toward a Semantic Toolset for Exploratory Programming

In the following, we describe another application of semantic conversations that aims at a broader adoption of semantic tools in exploratory programming systems. Our approach is based on the observation that many exploratory programming systems employ *object-oriented user interfaces*. An object-oriented user interface (OOUI) is a—predominantly graphical—type of user interface that employs an object-oriented metaphor and an injective mapping from (complex) domain objects to visual elements [11]. Examples of OOUIs can be found in several domains, including graphical editors such as Microsoft PowerPoint, project management software such as Jira, and also several programming environments such as Eclipse, Scratch, and Smalltalk systems [38].

We propose a simple mechanism for OOUI frameworks that integrates semantic object interfaces with the visual mapping of OOUIs. This allows users to select domain objects on their screen to talk to them in natural language. At the same time, domain developers are not required to put manual effort into writing prompts or preparing contextual information for LLMs, as the agent still fetches all required information by itself through automated experiments. For example, in a project management system that organizes task items in boards and lanes, a user could select a lane and ask for a semantic filtration or summary of the items it contains (fig. 8.7).

8. Applications

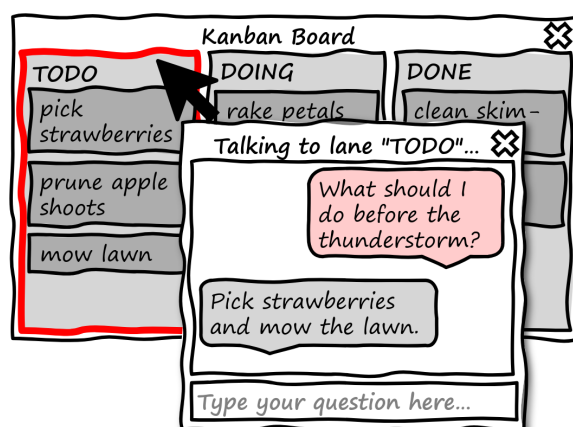


Figure 8.7.: Through the generic integration of semantic object interfaces into object-oriented user interfaces, users can select arbitrary domain objects from the UI and start a natural-language conversation with them. Here, a user talks to a lane in a project management system to filter its items.

We apply this concept to graphical, object-oriented programming systems such as Squeak/Smalltalk, where tools such as code browsers, (projectional) editors, (back-in-time) debuggers, and profilers represent views on underlying code objects and artifacts derived from them such as classes in packages, code blocks in methods, and call stacks or program traces. In this way, programmers can chat with code objects to ask for the responsibilities or collaborators of a class (fig. 8.9), explain or refactor a code block (fig. 8.10), search for the origin of values or cause of state changes in a program stack or slice (fig. 8.8), identify the bottlenecks in a program trace, and more.

Thus, we effectively upgrade existing programming tools to *semantic tools* by extending them with a conversational interface for an agent that will autonomously explore the programming artifacts shown in a tool. This allows programmers to express their questions and intents about programming artifacts in natural language and in the context of their exploratory session (which is captured in the conversation history of the agent). While the efficacy of this approach still hinges on the advancements in LLM capabilities and the precision of prompt engineering, this integration into exploratory programming systems promises a noticeable reduction in semantic distance, thereby supporting programmers' conceptual focus and enhancing their interaction with systems at a high level of abstraction.

8. Applications

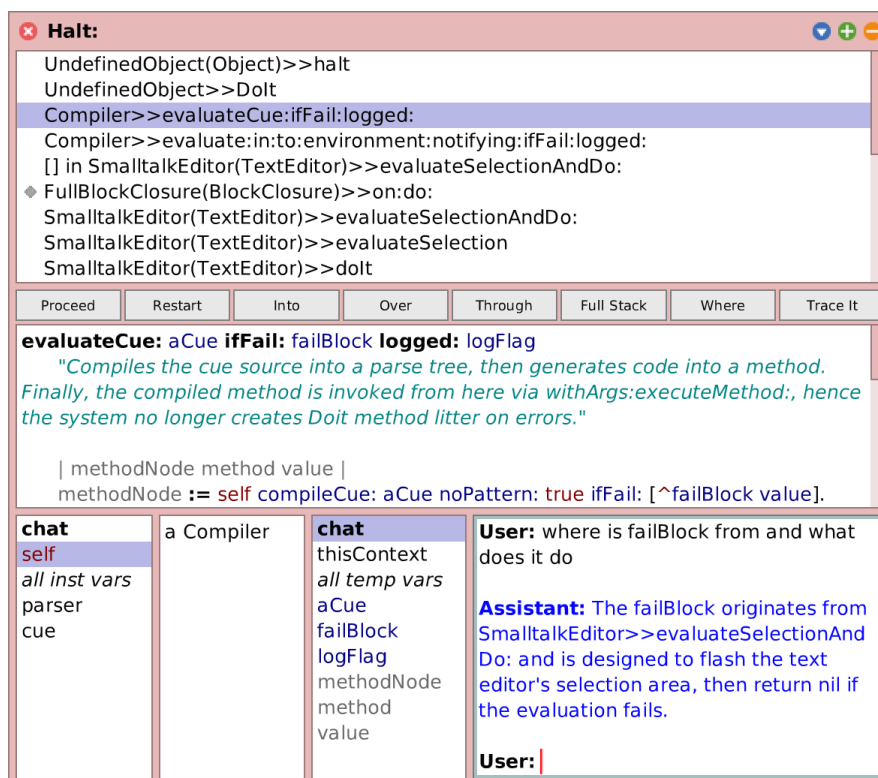


Figure 8.8.: In a semantic debugger (here: the Squeak debugger), programmers can ask for the origin and meaning of values on the program stack.

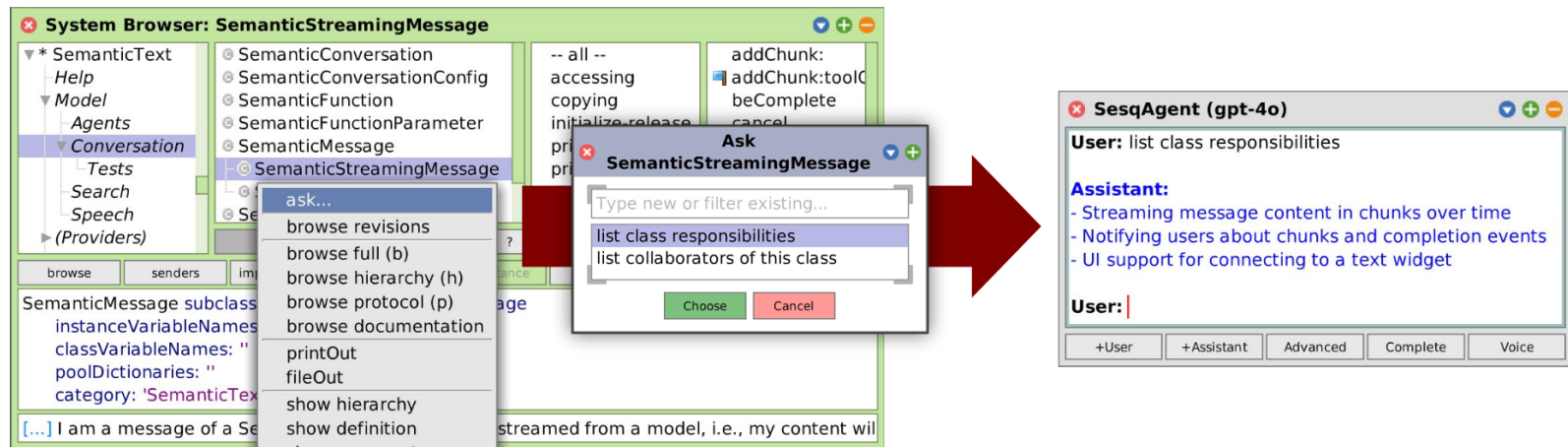


Figure 8.9.: In a semantic code browser (here: the default system browser of Squeak 6.1Alpha), programmers can engage in natural-language conversations with classes to explore them, for example, by asking for their responsibilities or collaborators.

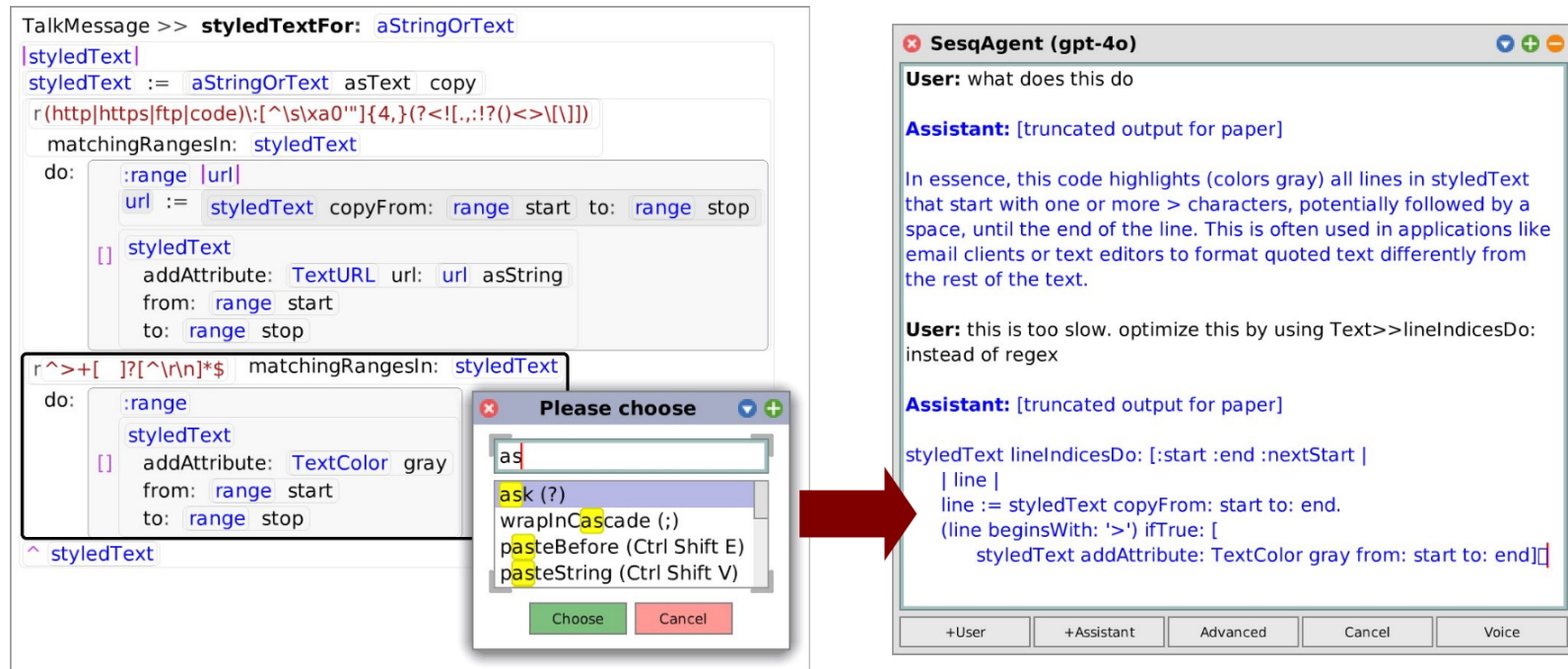


Figure 8.10.: In a semantic projectional editor (here: Sandblocks [6]), programmers can chat with single code blocks to explain, refactor, or execute them.

Chapter Summary

Through different case studies, we have demonstrated how programmers can use semantic suggestions and completions to create prototypes without conducting all experiments and research on their own, and how they can use semantic conversations to explore the state and behavior of objects through natural-language questions. We have illustrated that by generically enhancing existing programming tools with semantic object interfaces, we can allow programmers to access different artifacts in the programming systems in a more conceptual, high-level style.

9. Discussion

In this chapter, we discuss the opportunities and challenges of semantic interfaces for augmenting exploratory programming by reporting on our experience with our prototype of the semantic workspace. To this end, we examine the potential of semantic technologies for our applications in the semantic exploration kernel and evaluate their performance. We then discuss the impact of semantic interfaces on the experience of exploratory programmers. Finally, we address ethical considerations regarding the use of semantic technologies.

In [appendix C](#), we provide distilled recommendations for other tool developers who plan to integrate semantic technologies into programming systems.

9.1. Potential of Semantic Technologies

To deliver relevant suggestions and conversations for semantic tools, the semantic exploration kernel hinges on the capabilities of the underlying semantic technologies. Here, we provide insights on using language models and other suggestion strategies for powering the suggestion engine and the exploratory programming agent of our prototype.

Over the past six months, we have successfully used our prototype of semantic suggestions and semantic conversations for several tasks of a similar size to that of the examples in [chapter 8](#) and [figs. 7.1 to 7.3](#). At the same time, semantic interfaces have not yet become part of our daily used toolset, which we mainly trace back to the high monetary cost, limited accuracy, and noticeable delays of our current prototype as discussed below. Still, we believe that these observations are no fundamental limitations to our approach but can be improved through the onward evolution of faster and more powerful language models and systematic prompt engineering or fine-tuning.

Remark. This thesis presents our exploratory research on the integration of semantic technologies into programming systems. To test the general feasibility of our approach, we implemented a proof-of-concept prototype for central ideas. However, we did not make any systematic efforts such as tuning hyperparameters or prompts to improve the quality or performance of our semantic applications. Thus, any limitations noted in this discussion only refer to our prototype and do not necessarily indicate fundamental issues with the overall approach.

Semantic Retrieval

In our prototype of the suggestion engine, we implemented semantic retrieval for method and class artifacts. For semantic suggestions and ranking, our implementation employs document embeddings as well as term-based methods (TF-IDF and correlation search).

In our experiments, we mostly experienced suggestions from embedding-based search as helpful. For example, we could reuse suggested similar implementations in many situations, discovered some unexpected duplications during code browsing, and were able to understand documentation commentary that was scattered across multiple methods in context. However, the `text-embedding-3-large` model by OpenAI that we used offers limited capabilities for encoding abstract concepts of source code but overemphasizes specific vocabulary and document size. This impedes semantic similarity search and semantic ranking, in particular reducing the relevance and irredundancy of ranked suggestions. We believe that the fitness of embeddings could be improved by fine-tuning an embedding model exclusively for conceptual similarity, using ad-hoc fine-tuned text embeddings [104], or preprocessing documents with the help of another LLM prior to embedding them.

Being too sensitive to the document size, semantic search using embeddings also cannot be used to find related methods to an early unfinished draft of the programmer (see [page 48](#)). For this purpose, our TF-IDF search strategy yielded better results. However, as TF-IDF only compares isolated literals, it does not support synonymous or homonymous identifiers, causing false negatives (missing suggestions) for similar implementations that use different names and false positives for implementations that overload names. The same limitations apply to our (also term-based) correlation search. We believe that by representing individual literals using concept-aware embeddings that include their semantics or reference graph [31] instead of sparse TF-IDF embeddings, false negatives from synonyms could be reduced.

Semantic ranking using probabilistic sampling from clusters showed mixed results in our experiments. In general, this strategy balanced relevance, irredundancy, diversity, and representativeness better than other approaches such as top-k selection (see [section 5.3](#)). However, we noted an insufficient accuracy of relevance scores in many situations: first, relevance scores from semantic similarity search underrepresent conceptual relevance in the ranking as described above. Second, relevance scores from TF-IDF search are too imprecise for methods with few literals—for example, a method with a single selector that is also contained in the query will receive a perfect score of 100%. To create a global ranking of artifacts from different search strategies, their relevance scores need to be normalized to a common scale.

Exploratory Programming Agent

Due to their statistical foundations, LLMs and thus agents based on them are inherently subject to the risk of erring or failing. To err means to provide wrong answers, either caused by limited reasoning abilities or by the tendency of LLMs to hallucinate: for instance, our agent was unable in some cases to write executable scripts based on given source code or error messages, and it would occasionally send fictitious messages to objects. To fail means to terminate without finding a solution: sometimes, our agent would stop after conducting a couple of unsuccessful experiments, stating that further knowledge about a system would be required. In one example, the agent explicitly rejected the task of extracting an API key from the object graph of a version control system because it would be “not allowed to provide confidential information”.

Examples. To quantify the abilities of our current agent, a larger evaluation based on representative data from actual exploratory programming sessions would be required, which is outside the scope of this work. However, we can illustrate its current state through a few anecdotal examples.

For the questions about the Text object from [section 8.2](#), the agent delivered correct and useful answers in approximately 80% of all requests. In the remaining cases, it would fail to locate the required interface for retrieving all attributes of the text but only return the attributes for the first character; reinvent the wheel and spend many times tokens and seconds more manually accessing the underlying data; or suggest non-functional or idiosyncratic code snippets for the third question.

In the semantic debugger (see [fig. 8.8](#)), the agent would sometimes be lazy and only answer the first part of our question until we insisted.

We were able to research and apply different methods of Squeak’s graphics framework Morphic through the agent to create, alter, and decorate a graphical widget. When we requested to add an animation, the agent would in some cases generate the correct code but store it under the wrong hook; in one attempt, it added an erroneous method to the widget that caused the Morphic environment to crash.

On one occasion, we used the agent successfully to extract the daily activity on the squeak-dev mailing list^a and export it to a CSV file. When we asked it to render the results as a scatter plot manually (Squeak does not provide a built-in diagram framework at the time of writing), it made a conversion mistake that corrupted the x-coordinates of all points in the plot. After we asked it to fix the mistake, it entangled itself into a never-ending chain of thought, questioning basics of the

Smalltalk syntax, researching unrelated protocols, and modifying and complicating other parts of its scripts, before we stopped it after several minutes and \$10 spent (which significantly exceeds the usual costs of typical questions).

^aUsing Squeak Inbox Talk, see [fig. 7.3](#).

In general, the agent operated most competently for smaller tasks related to popular domains. However, in most situations, we could send follow-up messages to the conversation to retrieve the correct answer by refining the task, telling the agent what it did wrong, and providing it guidance through the solution with a varying degree of detail.

Several weaknesses of our agent are due to two limited capabilities of the GPT-4o model we used: first, this model showed a limited proficiency regarding the Smalltalk programming language and standard libraries, and it often mixed up protocols from different Smalltalk dialects such as Squeak and Pharo. Second and more importantly, despite our prompt, the model only applied the exploratory paradigm to a limited degree and was not eager enough to consider and try out a larger number of different classes and messages. We believe that by training a model on an extended set of Squeak/Smalltalk source code and fine-tuning it for exploratory practices, both these capabilities could be further improved.¹ Alternatively, even a more systematic approach to prompt engineering with extended chain-of-thought or few-shot prompts might improve the agent's behavior.

The limited language proficiency of the model was also a critical limitation when generating semantic completions: despite being presented extensive examples of the syntax and relevant interfaces, the agent produced invalid completions in 50%–75% of all cases. While we could filter out these errors by dynamically testing completions, this reduces the quantity and the effective performance of suggestions substantially. Completions generally exhibited an acceptable diversity over the different provided examples even with a naive prompt design. Still, by chaining multiple prompts or increasing the temperature of the model, it would be possible to generate more diverse completions while sacrificing the performance or correctness of results. During stage-2 generations, the agent usually provided satisfactory results for recontextualizing completions to a similar user input even when using the less powerful model GPT-4o mini; only if the distance between stage-1 generations and user

¹For comparison, ChatGPT's code interpreter shows a higher proficiency in writing Python code for computational notebooks, using proprietary libraries to summarize documents, and correcting its own coding errors. We argue that these are similar and trainable traits to the typical system interactions in exploratory programming.

9. Discussion

Semantic interface	Response time	Monetary cost per query	Monetary cost per hour
Semantic suggestions	0.5 s – 0.9 s	\$0.0001 – \$0.001	\$0.08
Semantic completions			
Stage-1 generations	10 s – 15 s	\$0.15	\$18
Stage-2 generations	1.1 s – 1.3 s	\$0.0005	\$1.8
Total	11 s – 16.3 s		\$20
Semantic conversations			
Simple to moderate tasks	2 s – 4 s	\$0.1 – \$0.5	\$1 – \$60
Complex tasks	5 s – 10 s	\$0.5 – \$5	\$5 – \$30

Table 9.1.: Main response times and memory consumptions for the tools in our prototype of the semantic workspace. Scattering of values is caused by the varying complexity of tasks, different suggestion strategies employed, and the uncertain frequency with which programmers may use tools.

inputs grew too large (because we did not re-run stage-1 generations frequently enough), the model would tend to hallucinate fictitious names again.

9.2. Performance of Semantic Tools

In our setting with cloud-based language models and local vector stores, resource consumption mainly manifests as response times in semantic retrieval and the exploratory programming agent, memory consumption for semantic corpora (vector stores), and monetary costs for using language model services. While memory consumption did not exceed 100 MB in our experiments, response times and monetary costs of our prototype indicate that semantic completions and conversations demand further optimizations (table 9.1).

Response Times

Semantic retrieval For semantic retrieval, response times occur in two situations: searching similar artifacts and embedding new artifacts in a semantic corpus. While our implementation of similarity search in the SEMANTICTEXT framework currently employs a linear in-memory search and does not use any indexes, it already performs fast enough for most use cases. We conducted our experiments in a Squeak 6.1Alpha image without additional packages installed² which contains about 65 000 methods and 3000 classes. Here, search-

²Testing environment: Squeak6.1alpha #23073 64-bit, SistaV1; OpenSmalltalk VM 20231218-1441 (2.5G bytecodes/sec, 170M sends/sec); Ubuntu 22.04 (virtualized in VirtualBox); AMD Ryzen 7 2700X @ 3.7 GHz.

9. Discussion

ing for the $k = 100$ most similar methods to an existing method took not longer than 150 milliseconds; searching for similar classes took about 10 milliseconds.

When performing a semantic search with an arbitrary query such as the contents of a workspace, the query needs to be embedded first, adding a latency between 400 milliseconds for short scripts (1 SLOC³) and 750 milliseconds for long scripts (200 SLOC) in our experiments. Analogously, embedding new or changed methods and classes in a corpus involves similar latencies, but these are performed in the background and are usually invisible to programmers. After changing larger amounts of code, such as when installing a new package, we can embed many documents in a single API request and parallelize requests to balance elastic scaling and possible rate limits. For example, in our (not perfectly balanced) prototype implementation, embedding the full Collections package of Squeak (about 3100 classes and 1900 SLOC) takes 7 seconds in the background; embedding all methods in the standard image (when installing our prototype initially) takes about three minutes.

Our implementation of TF-IDF search for similar methods takes about 15 milliseconds per literal of the query method (correlation strength: $R^2 = 0.99$) in the standard image. Since 75% of methods in the standard image contain not more than 6 literals, most queries will take fewer than 100 milliseconds; 99% of methods contain not more than 34 literals and can be queried for similar methods within less than 900 milliseconds. However, a few methods may contain more than 500 literals (usually “data methods” that contain serialized objects such as images or sounds), causing query times of 10 seconds or more.

Correlation search and ranking typically consume 2–5 milliseconds only. In total, a full pass of creating semantic suggestions for an open workspace typically requires between 500 and 700 milliseconds. To reduce these latencies, we could use local, smaller embedding models and maintain indices for semantic and TF-IDF search.

Exploratory programming agent Generating semantic completions took between 10 and 15 seconds in our experiments, which is mainly caused by the large context we pass to the exploratory programming agent. By optimizing the prompt, varying the amount of contextual information depending on the complexity of the problem, or pre-training an LLM for Smalltalk code completions, these latencies could be reduced. While it would be possible to stream completions similarly to agent responses in semantic conversations, we generally cannot validate incomplete expressions or preview their results, which would hinder effective filtering of noise in the completion menu. Stage-2 generations for recontextualization took between 1.1 and 1.3 seconds in our experiments due to their lower complexity. This makes it realistic to

³SLOC: Source lines of code (excluding empty lines).

recontextualize completions whenever programmers add or change a few words; however, for a higher interactivity, it would be necessary to employ even smaller, preferably local models for this stage or complement LLM-based recontextualization with non-semantic heuristics as proposed in [section 5.4](#).

Response times for semantic conversations largely depend on the amount of context provided, such as previous messages in the conversation and required experiments: for simple questions (e.g., the first two messages in [fig. 8.5](#)), response times usually vary between 2 and 4 seconds, while complex questions that involve sequential experiments (especially those caused by internal trial and error of the agent) can result in response times between 5 to 15 seconds or more in a few cases. By streaming responses character-wise from the API, we reduce the delay before the start of the answer by factors between 20% and 80%.

Memory Consumption

Maintaining embeddings for larger corpora of documents can require substantial amounts of storage. In the current implementation of `SEMANTICTEXT`, we store embeddings in the image memory and do not maintain additional indexes (but instead perform a linear search). To balance memory consumption against precision, we shorten all document embeddings of OpenAI's `text-embedding-3-large` model (which supports truncating embedding vectors through Matryoshka Representation Learning [91]) from 3072 to 256 dimensions (each encoded as a 32-bit float). This reduces the accuracy of semantic retrieval compared to the full embedding size by an average loss in relevance scores of about ± 0.05 .⁴

Using this configuration, a semantic corpus of all methods in the standard image consumes about 90 MB, and a corpus of all classes consumes about 4 MB. It would be possible to further improve memory consumption while trading off accuracy or speed by further reducing embedding sizes or offloading embeddings to disk (and maintaining an index for them in the image memory). Alternatively, training a smaller embedding model for our particular use case of conceptual similarity of Smalltalk code could allow to reduce memory consumption while maintaining a high accuracy.

⁴Christoph Thiede and Curt Kennedy: Discussion on the quality loss in similarity search when shortening embeddings. In: *It Looks Like text-embedding-3 Embeddings Are Truncated/Scaled Versions from Higher Dim Version*. OpenAI Developer Forum, 2024-02-07. URL: https://web.archive.org/web/20240413011050/https://community.openai.com/t/it-looks-like-text-embedding-3-embeddings-are-truncated-scaled-versions-from-higher-dim-version/602276#post_14.

Monetary Cost

When using a cloud embedding model, monetary cost for semantic suggestions occurs when updating semantic corpora or embedding inputs such as current workspace contents for querying corpora. Using OpenAI’s text-embedding-3-large model, embedding all methods and classes in the standard image costs approximately \$0.85 (US dollars). During our experiments within the last six months, regular code changes that we performed ourselves or installed as updates of the Squeak Trunk image accounted for less than \$0.05 per month. Embedding an average method draft or workspace script for querying averaged to 0.01 ¢ (US cents), and even long scripts (200 SLOC) never exceeded costs of 0.1 ¢. Assuming a programmer would type continuously into a workspace and we would update suggestions every 5 seconds, this would generate costs of fewer than \$0.08 per hour.

Generating semantic completions is generally more expensive than creating suggestions because many generative language models require more resources than embedding models. Using our unoptimized prompt design, generating $k = 10$ stage-1 completions costs about 15 ¢ and generating stage-2 completions for them costs about 0.05 ¢. If a programmer typed continuously into a workspace, we could update stage-1 completions every 30 seconds and recontextualize them every second for a price of \$0.08 (for suggestions, see above) + \$18 + \$1.8 \approx \$20 per hour. These costs would be considered impractical in most settings and emphasize the need for optimizing completions through prompt tuning or smaller, fine-tuned models as noted above.

Monetary costs for semantic conversations vary depending on the complexity of questions and answers, ranging from \$0.10 to \$0.50 for simple questions but rising to several dollars when encountering a lot of trial and error for more challenging questions. For a programmer who adopts semantic object interfaces in their everyday toolset, this would involve expenses of at least \$5 per hour even when estimating the number of questions a programmer expresses very conservatively as 10 per hour [26]; following Jevon’s paradox, the actual cost could be many times higher as programmers might become negligent of their expenses.⁵

*

Considering the ongoing trend of more powerful, faster, and less energy-consuming language models being developed, we assume that the latencies

⁵We also experimented with OpenAI’s gpt-3.5-turbo model instead of gpt-4o-2024-05-13. While this would improve response times and reduce costs by 5 to 10 times, GPT-3.5 models are noticeably less capable, requiring us to ask questions five times or more (due to the non-deterministic behavior of LLMs) before we would get a helpful answer, if at all.

and expenses for invoking the suggestion engine and the exploratory programming agent could become small enough to no longer pose a practical barrier for programmers and organizations with more affluent budgets.⁶ For illustration, during the last twelve months, OpenAI's respective flagship LLM (from gpt-4-0613 to gpt-4o-2024-08-06) has improved by a factor of at least two in speed and by a factor of six in monetary cost. At the same time, we see *small language models* as a promising alternative to cloud-based models, as they could be trained to handle particular tasks such as generating Smalltalk code completions or engaging in exploratory programming competently and efficiently and could be run on commodity hardware with reduced response times [94].

9.3. Exploratory Programming Experience

In the following, we discuss how semantic tools can support exploratory programmers in maintaining their workflow. For this, we analyze the main influences of the semantic workspace on the experienced immediacy of programmers [74]: how well it supports programmers in the hierarchical research process, how closely it is integrated in the surrounding programming system, and how existing performance limitations may disrupt programmers in their flow.

Improved Support of the Research Process

We propose semantic interfaces for programming in this thesis because we intend to assist programmers in handling the—typically hierarchical—research process of common exploratory practices and thus improve their experience of semantic immediacy. This experience crucially depends on the quality of cooperation between programmers and the semantic programming system. The quality of cooperation consists of effective communication, division of labor, and trust in the system.

Level of abstraction Semantic interfaces allow programmers to express their questions on higher abstraction levels while delegating lower-level experiments to the exploratory programming system. Thus, they help to bridge the gulfs of execution and evaluation to the software system [37], reduce distractions and cognitive load in programmers, and support them in maintaining their

⁶We acknowledge the high variance in budgets worldwide. For example, a senior developer in Silicon Valley with a three-digit hourly wage might regularly afford semantic conversations or even semantic completions today, while an entry-level programmer in a developing economy who earns less than \$1 per hour could likely not even afford semantic suggestions.

flow [13]. Furthermore, this abstraction can facilitate the learning curve for programmers who are exploring an unknown system, and we believe that it could also support programming novices in their educational process by separating domain concepts from programming language concepts.

On the other hand, the streamlined exploration process can also result in some kind of “tunnel vision” for programmers. For example, while browsing a class to find protocols for a certain subproblem in a manual research process, programmers might encounter other protocols by chance that give them a peripheral impression of a system’s domain concepts and capabilities, or they even might find an unanticipated method that directly solves their overarching problem. As semantic tools can automatically answer those questions, they reduce such manual experiments, and programmers will make fewer serendipitous discoveries.

Natural language interfaces Semantic interfaces allow programmers to communicate questions and intentions without translating them into a formal (programming) language, just by expressing them “as they come through their mind”. For example, instead of typing `DateAndTime fromUnixTime: anOrder creationDate / 1000` into an inspector or editor, a programmer can type (or even speak⁷) “when was the order filed?” into a chat interface or as a comment into their code. To ask follow-up questions, programmers do not have to repeat or modify existing inputs but can express them in the semantic context of an existing conversation.

Analogously, answers of semantic conversations are provided in natural language that programmers usually do not have to “decode” into familiar concepts. Finally, answers can be customized and personalized based on the intentions and preferences of programmers: for example, novice programmers could ask the exploratory programming agent to explain its proposed solutions step-by-step, while experts might prefer concise outputs that align closer with their own mental model. Thus, semantic interfaces reduce the gulfs of execution and evaluation and contribute to the semantic immediacy of programmers.

At the same time, natural language interfaces can remove incentives for programmers to precisely explicate their thoughts, as conversational agents are rather good at resolving uncertainties and ambiguities in questions autonomously. However, explication of thought is a critical part of common problem-solving strategies. While programmers will usually wish to delegate a problem to the exploratory programming system when they use a semantic interface, explicating particular questions can also help them better understand overarching questions and problems. So, by commonly relying on vague

⁷SEMANTICTEXT supports experimental voice conversations, which can further help to bridge semantic distances in human-computer interaction.

natural language questions, programmers may impede their own research activities in the short or long term.

Example. A programmer might ask “when an order was filed” while they are creating a user interface for a shop system. Only when they are forced to manually translate “filed” into technical terms of the order interface—such as `creationDate`, `submissionDate`, or `receiptDate`—might they realize that there are multiple options and start thinking about what information exactly they want to display to users. If the programming system is tasked to answer this question, it may lack the overarching context of the intended user interface and make a decision on its own that does not necessarily align with the—yet undeveloped—goals of the programmer.

Delegation of control When programmers remain on a conceptual level of abstraction, they have to delegate control to another instance, namely the exploratory programming system. If responsibilities are clearly separated and temporal immediacy is maintained, this can be an effective division of labor. However, as contemporary semantic technologies—and thus semantic tools based on them—are still prone to provide incorrect or no answers, programmers’ trust in these tools might diminish. If semantic tools err or fail or programmers suspect them thereof, semantic abstractions become leaky [58, chap. 26]: programmers have to intervene and fall back to traditional low-level practices and experiments. During this, they need to either understand the previous attempts of the system and correct them or discard the system’s work and make most of the effort again.

Realizing that trust in current semantic technologies is limited, we emphasize that the behavior of semantic programming tools must be explainable, so that programmers can verify answers or build upon their experiments and developers of semantic tools can debug problems. For semantic suggestions and completions, we have implemented a provenance mechanism in the suggestion engine, which allows programmers to understand the context of suggestions (see page 38). For semantic conversations, we have included an option for displaying all experiments and thought processes of the exploratory agent. Despite these approaches, explaining the representations of large embedding models for semantic search and the decisions and attention of generative LLMs for autonomous agents are evolving challenges [79; 112, sec. 8.1.3], and large parts of the semantic workspace effectively remain gray boxes. In addition, even when the internal steps of the system are explained conceptually, we acknowledge the intrinsic complexity programmers face when switching between abstraction levels to understand these concepts.

Level of support In our concept of the semantic workspace, we explored different gradations between augmentation and automation for semantic tools, both of which intend different cooperation styles between the programmer and the system.

Automation interfaces are designed for programmers to maintain a high-level flow and avoid descending into subordinate research processes. When tools meet these expectations, this can substantially reduce distractions in programmers, streamline their workflow, and make them more efficient, leading to an improved semantic programming experience. However, for every interaction with a high-level semantic tool, programmers are required to invest some effort by explicating their questions or intentions and using a separate interface. If the tool fails to deliver a helpful answer due to the limitations discussed earlier, the investment is lost and the expectations of programmers are not met, exposing the semantic interface as a leaky abstraction. Then, the flow of programmers is broken as they have to switch their approach and handle lower-level details such as browsing protocols and writing code manually. Even when the accuracy of semantic technologies evolves in the future, the law of leaky abstraction suggests that the success of semantic tools will remain uncertain.

On the other hand, augmentation interfaces do not involve a fundamental change in the workflow of programmers but allow them to retain their familiar methods. In this setting, suggestions by semantic tools are rather optional and not strictly expected, making their failures less drastic or disruptive. For example, suggestions for similar methods or code completions only provide selective inspirations for programmers but are never exhaustive, so programmers will still commonly conduct their own research beyond the provided recommendations. In addition, programmers may actively seek to engage in subordinate research processes on their own for educational purposes or intrinsic motivation to low-level coding [65].

Integration into Programming System

In our prototype of the semantic workspace, we attempt to minimize spatial and semantic distances by integrating semantic tools as closely as possible into the existing programming system. To this end, we integrate semantic conversations directly into object inspectors and other exploration tools, so programmers can ask semantic questions as easily as they can select traditional elements (such as variables or menu commands) in these tools. Semantic messages further reduce spatial and semantic distances when programmers are currently interacting with objects through scripts, since they allow asking high-level questions through the same scripting interface.

9. Discussion

We place semantic completions into the autocompletion menu of workspaces, code browsers, and related tools to assure that they are not farther away than a few lines from the text cursor of the programmer, thus minimizing spatial distances.

On the other hand, semantic suggestions are more comprehensive, so we collect them in a suggestion space that is located in a separate docking bar at the edge of the screen and thus more distant to the current focus window of the programmer. This reduces the chance of programmers discovering relevant suggestions unless they are actively looking at the suggestion space. However, this placement is our trade-off to avoid disrupting programmers with many possibly irrelevant suggestions closer to their cursor. Finally, the traditional equivalent to the suggestion space—opening multiple code browsers or message traces—involves even larger spatial distances due to the typical layout and size of these tools.

Performance Limitations

A noticeable resource consumption can impede the programming experience in different ways: large response times can reduce the experience of temporal immediacy, but programmers usually accept delays of up to 4 seconds for common tasks, though delays below 1–2 seconds are preferred for frequent tasks [56, p. 473]. These thresholds are usually met by our prototypes for semantic completions and simple semantic suggestions. However, the delays of semantic completions and some semantic conversations for complex questions might challenge the patience of programmers, until LLMs are further accelerated.

Beyond the three conventional dimensions of immediacy proposed by [74], we observed a fourth kind of “psychological distance” when dealing with monetary costs in our experiments. When using semantic completions and conversations, we were constantly afraid of causing expenses (“How expensive will this completion be? Could that answer really be worth two dollars?”), which frequently kept us from using them more frequently, even though we did not exhaust our available research budget. We have tried to mitigate this effect by displaying a prominent expense watcher in the programming system and defining quotas for each semantic question (see section A.3). Still, we couldn’t completely shake off the uneasy feeling whenever we wished to complete a statement or talk to an object.

9.4. Ethical Considerations of Semantic Technologies

We are concerned about several ethical issues regarding the use of (cloud-based) language models in exploratory programming systems. These include the concentration of economic and political power, high energy intake and water consumption (a typical question to our exploratory programming agent might use 0.05 kWh and between 50 ml and 750 ml water [93]⁸), and the training of models based on the intellectual property of unconsulted content creators and labor of inadequately provided click workers⁹. Language models can also exhibit potentially unsafe biases, which are poorly understood and could impact the accessibility, internationalization, or safety of data analyses or recommended solutions [99].

Before adopting semantic programming tools in practice, we advise to consider these implications and evaluate possible measures. For example, tool developers should favor language models that were trained on fair and open-source datasets, run them locally if possible (e.g., in the case of small language models), or host them in trusted and ecological compute clusters. More generally, we emphasize the need for future research on explainable and efficient language models, evaluation and mitigation of biases, and political regulations regarding training and operation of language models.

Chapter Summary

Semantic interfaces show strong promise for improving the programming experience in our experiments by streamlining the exploratory research process. However, programmers could miss opportunities for learning and decision-making when overusing high-level support tools, and they might not trust semantic tools that fail too often and do not explain their answers. To improve the accuracy and correctness of semantic tools, tool developers should invest in better training and prompting of language models, particularly for conceptual understanding of source code and exploratory programming practices. Similarly, high response times, monetary costs, and environmental and social impacts of current LLMs prevent programmers from using semantic completions and conversations easily and frequently, so further work is required to tune prompts, optimize models, and ideally scale them down for sustainable on-device usage.

⁸Actual numbers might vary because the data in [93] refers to the older model GPT-3, depends on the region of compute clusters, and relies on ambiguous estimates by Microsoft.

⁹Billy Perrigo. 2023-01-18. *OpenAI Used Kenyan Workers on Less Than \$2 per Hour to Make ChatGPT Less Toxic*. Time. URL: <https://web.archive.org/web/20240704034409/https://time.com/6247678/openai-chatgpt-kenya-workers/>.

10. Related Work

In this chapter, we provide an overview of related work in the areas of suggestion mechanisms for programming systems and high-level programming interfaces. This complements our introduction of exploratory programming practices, tools, and semantic technologies presented in [chapter 2](#).

Our concept of a semantic exploratory programming system to which a programmer can delegate tasks shares similarities to a pair-programming setup [5], where a navigator gives directions to a driver, the driver executes them, and the navigator reviews the results. Several approaches have been proposed to mimic the role of the driver through programming tools. Still, to our knowledge, the semantic workspace is the first work to enable conceptual collaboration of exploratory programmers and an intelligent programming system with a variable degree of automation versus augmentation.

10.1. Suggestions in Programming Systems

Traditionally, programmers can search source code through code browsers or other tools that retrieve methods based on call graphs [16, chap. 10; 25, sec. 6.2; 67], semantic similarity [89], or runtime behavior [67, sec. 1.8]. To improve the discoverability of relevant solutions, many approaches have been proposed that provide programmers with proactive suggestions of existing or new code.

One of the most established recommender systems for source code is traditional code completion tools, which are integrated in many programming systems and usually suggest single contextually relevant identifiers while programmers are typing code. To make contextually relevant suggestions, they typically incorporate the call graph or usage statistics of software systems [69], typing and runtime information, or previous changes of programmers [48]. Examples of traditional code completion tools include Microsoft IntelliSense in Visual Studio and Visual Studio Code¹ as well as the `AUTOCOMPLETION` package for Squeak we used in our prototype. Other tools also suggest possible methods for exploratory code browsing [7, 50] or even use recommender systems to assist programmers in navigating through complex user interfaces of programming systems [35].

¹<https://code.visualstudio.com/docs/editor/intellisense>

Similarly to code completion tools, visual programming editors such as ETOYS or SCRATCH [47] offer *palettes* through which programmers can list and apply different relevant methods for their programs.

Beyond recommending existing source code, contemporary suggestion tools also offer new code that they synthesize with the help of LLMs based on the existing code context. Popular tools of this kind such as GitHub Copilot², Tabnine³ and IntelliCode Compose [59] provide code completions through two different interfaces: either they insert a *ghost text* in the editor after the existing code typed by programmers, or they display a separate suggestion pane, from where programmers can compare multiple completions and insert them in the editor. Most programmers use generated code completions to accelerate their typing and to explore different interfaces and solution approaches [4]. Other code suggestion tools also recommend code changes to fix bugs, adhere to coding styles (often based on the annotations of linters), or continue refactorings that programmers have begun [75].

10.2. High-Level Programming Interfaces

While suggestion tools usually contribute small portions of work to programmers, high-level interfaces provide new means for programmers through which they can explicitly delegate tasks to particular tools.

Question-based debugging tools Different tools have been proposed to support programmers during debugging sessions by answering high-level questions [14, 36]. For example, the WHYLINE approach allows programmers to ask questions about the causes of certain events in their program [24]. While these questions are limited to a rigid box of building blocks that can be combined to form queries of predefined patterns, this approach already abstracts away from low-level interactions and queries to the system [39].

CHATDBG provides an LLM-based agent to answer natural-language questions about an errored program in a debugger [28]. It automatically conducts small experiments by inspecting different parts of the stack and executing scripts in the program context to identify the root causes of errors and recommend possible fixes.

Natural-language prototyping tools Tools such as SPELLBURST [3] and OPENUI⁴ facilitate iterative prototyping of visualizations and user interfaces by allowing programmers to provide natural-language instructions. SPELL-

²<https://github.com/features/copilot>

³<https://www.tabnine.com/>

⁴<https://github.com/wandb/openui>

BURST also offers a node-based visual programming interface for managing derivations and alternatives of prototypes.

Natural-language programming interfaces The vision of *natural-language programming* is to enable programmers to write entire programs in natural language and refrain from technical details. Past approaches to this vision employ rule-based grammatical heuristics or LLMs to translate natural-language descriptions into code [34]. Different metaphors such as tools⁵ and operating systems [33] were discussed to structure natural-language directives and scale them to larger no-code applications.

NAVĀ is an extension to traditional programming languages that allows for finding and invoking methods on components through declarative queries by using an extensive ontology of domain-specific, programming-related, and common-sense knowledge [53]. While component developers and users are still required to express intentions in a formal language to maintain and access detailed specifications, this approach enables programmers to communicate with software systems in a semantic, interface-agnostic style.

Conversational programming agents Conversational agents such as GitHub Copilot Chat, Tabnine Chat, and Sourcegraph Cody [18] allow programmers to ask questions across a wide range of topics: programmers can chat with them in natural language to explore and understand code bases or to learn about other libraries and general programming practices; ask them to explain and fix errors in their code; or request refactorings or additions to their code [27, 49, 51]. However, these agents typically only have access to the static context of a code base and are unable to conduct experiments or research about other packages on their own.

Chapter Summary

Existing suggestion tools recommend identifiers, methods, or generated code snippets that programmers can apply to their program or use for further exploration. Other high-level programming interfaces provide conceptual, often natural-language interfaces, through which programmers can automate smaller search and programming tasks to facilitate programming activities such as refactoring, prototyping, and debugging.

⁵GPTSCRIPT: <https://github.com/gptscript-ai/gptscript>

11. Conclusion

At the beginning of this work, we have asked the following question:

How can we augment the exploratory programming workflow by integrating semantic technologies into exploratory programming systems?

To answer this question, we have started by investigating the nature of exploratory programming and modeling it as a *hierarchical research workflow* of recursive questions and answers ([chapter 2](#)). Based on this model, we have identified two major challenges: large semantic distances and information overload, which arise from handling questions and experiments at different abstraction levels and which disrupt programmers in their flow. We have introduced *semantic technologies* as a possible remedy: semantic retrieval, which embeds domain artifacts into a numeric space and allows to search and compare them based on their position and distance in that space, and LLMs, which can process and generate unstructured text and code, conduct conversations with human beings, and solve problems through inner monologue.

To address the challenges of exploratory programmers, we have proposed our model of the *augmented exploratory programming workflow*, in which semantic technologies are integrated into programming systems to support programmers at different stages of their research ([chapter 3](#)). In our concept of the *semantic workspace*, we have proposed three different tools for such integrations along a level-of-support spectrum ranging from augmentation to automation interfaces: (i) *semantic suggestions*, for which the system follows the steps and experiments of programmers and suggests new artifacts or experiments such as methods to browse; (ii) *semantic completions*, which continue the plans of programmers through contextually generated text insertions; and (iii) *semantic conversations*, through which programmers can ask conceptual questions about objects by using a natural-language chat interface or an extension to the programming language.

We have described our design of a *semantic exploration kernel* ([chapter 4](#)) to harness semantic technologies for programming tools through two main components: a *suggestion engine*, which collects artifacts and experiments of programmers in our blackboard framework and defines different retrieval strategies for searching and recommending further artifacts; and an *exploratory programming agent*, which employs LLMs to plan and conduct experiments,

11. Conclusion

interacts with the system through technical interfaces and generated code, and communicates with programmers through high-level interfaces. To create suggestions, we have explored different approaches for finding *similar* and *correlated* artifacts based on semantic embeddings and call graphs; and we have compared different ranking methods such as top-k selection, probabilistic sampling, and clustering to optimize for the relevance, diversity, irredundancy, and representativeness of results, finding that *probabilistic sampling from clusters* balanced all objectives most effectively (chapter 5). To construct a viable exploratory programming agent based on the GPT-4o model, we have defined behavioral policies through prompts and designed system functions for automated experiments (chapter 6). Finally, we have implemented these ideas in a prototype for the Squeak/Smalltalk environment by using our SEMANTICTEXT framework and the OpenAI API (chapter 7).

In our experiments (chapter 8), we have successfully used the semantic workspace to prototype a UI with the help of semantic suggestions and completions for available protocols and usage patterns of the UI framework, as well as to understand the design and usage of Squeak’s text-formatting package by having a semantic conversation with an example object. Beyond that, we have illustrated how semantic object interfaces can be integrated into further programming tools such as code browsers and debuggers to broaden our vision of a semantic exploratory programming system. While our preliminary experiences were promising, semantic tools currently produce many irrelevant or incorrect outputs due to the low accuracy of semantic technologies, and semantic completions and conversations are too expensive, unsustainable, and slow to use them frequently and interactively (chapter 9). Despite that, we think that semantic tools have much potential for improving the exploratory programming experience, since they allow programmers to delegate flexible parts of their research process and thus better maintain their high-level flow.

*

In the future, we want to pinpoint the systematic limitations of our approach by investing efforts in tuning our prompts and hyperparameters as well as fine-tuning or down-scaling language models. To this end, we will also consider compiling datasets for training and evaluating models by mining and annotating data from real exploratory programming sessions [2].¹

¹The fact that contemporary LLMs are unfamiliar with exploratory programming also suggests that they have seen few examples of this practice during training. This leads us to another insight: we appeal to the exploratory programming community to put more effort in documenting their exploratory workflows and practices publicly. Not only could this help to make exploratory programming more popular amongst developers but also serve as training data for future LLMs.

11. Conclusion

Another opportunity would be the conduction of a quantitative user study to determine the optimal degree of augmentation versus automation tools for maximizing the productivity and experience of exploratory programmers. Within this study, we could also attempt to identify types of semantic interfaces that programmers prefer for delegating parts of their work to the system.

Finally, we want to explore further means for enabling and fostering a broader collaboration of programmers and semantic exploratory programming systems. On the one hand, this could be achieved by tracking more exploratory activities of programmers to capture their context for semantic tools. On the other hand, we want to improve means for keeping programmers “in the loop” while semantic agents conduct research on their own. This involves finding better ways to explain the results and steps of semantic technologies, including technical model interpretability [79] and tools or visualizations for human-readable explanations. Furthermore, we want to define scope policies for the experiments and decisions of agents and establish an explicit channel for inquiries from systems back to programmers. By integrating semantic suggestions into different programming tools—such as relevance-based ranking in code search tools and contextual stack-trace filters in (omniscient) debuggers—, we also want to expand fine-grained, proactive support of semantic technologies in exploratory programming systems. Ultimately, we envision a form of semantic exploratory programming in which the system continuously “thinks along” the actions of a programmer (like a helpful pair programmer) by anticipating their intentions, exploring different ways to fulfill them through deep reasoning and research, and offering distilled results to allow programmers stay in their flow.

*

Much about the future of programming in an era of evolving AI technologies and possibly artificial general intelligence [78] remains uncertain [65]. While it is likely that low-level programming tasks will be further automated, analyzing domains and conceptualizing possible solutions have proven to be two of the hardest—and possibly hardest to automate—parts of software engineering. We believe that on the journey toward a “generational shift” in programming [55] and beyond, semantic tools can play a crucial role in supporting programmers to focus on high-level problems while communicating with running systems in a more natural way for an expedient, exploratory programming practice.

A. Accessing Semantic Technologies through the SEMANTICTEXT Framework

In this appendix, we describe our design and implementation of the SEMANTICTEXT framework for accessing semantic technologies in Squeak. SEMANTICTEXT provides an object-oriented domain model for semantic retrieval using document embeddings and for text generation and machine reasoning using conversational agents.¹ It also implements a client for relevant language models of the OpenAI API. Finally, it offers several tools to support prototyping of semantic applications.

In the following, we describe the three central parts of the framework: domain model, providers for different language models, and tool support.

A.1. Domain Model

We describe our domain model of semantic technologies for programming semantic retrieval systems and conversational agents.

Semantic Retrieval

To model semantic retrieval in the context of the Squeak ecosystem, SEMANTICTEXT defines the central class `SemanticCorpus` as a specialization of the `Set` class from the `Collections` package (fig. A.1). This class extends the responsibilities of a regular collection for containing any number of (domain) objects with those of a vector store for searching and filtering objects based on document embeddings.

For each object, the corpus computes and stores an embedding vector that is represented by a `FloatArray`, allowing for efficient comparison, memory consumption, and serialization. To compute embeddings, the corpus is provided with an `EmbeddingModel` (which may be implemented by different providers, see section A.2) and an `EmbeddingConfig` for controlling the embedding properties such as the length of embedding vectors. To embed non-textual object

¹<https://github.com/hpi-swa-lab/Squeak-SemanticText>

A. Accessing Semantic Technologies through the SEMANTICTEXT Framework

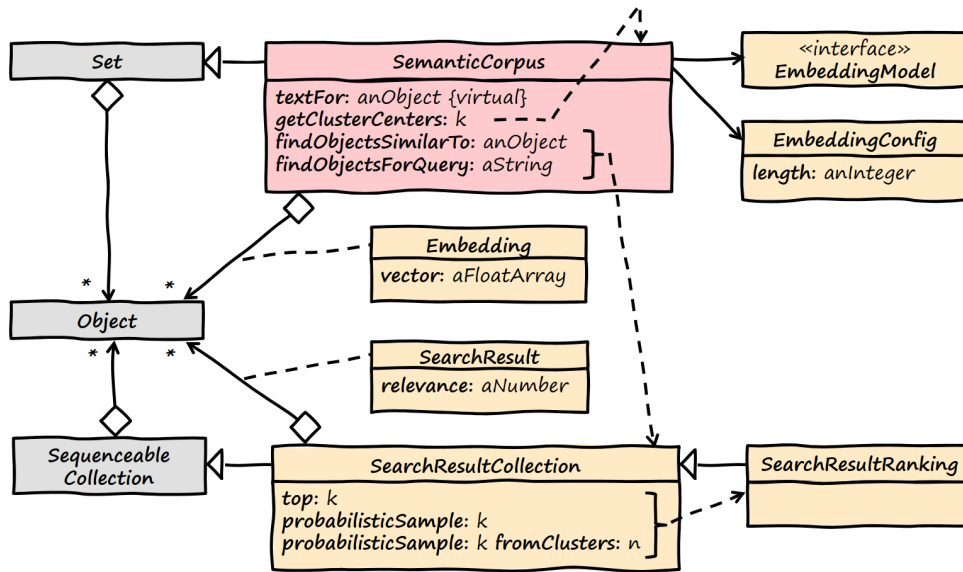


Figure A.1.: The general object model for semantic retrieval in SEMANTICTEXT (UML class diagram with Smalltalk-styled method signatures; default multiplicities are 1; dashed arrows indicate the return class of message sends). Programmers can use the entry class `SemanticCorpus` (red) to index domain objects with an embedding model and search, rank, or cluster them (gray classes are provided by the Squak base system).

elements, the corpus can be provided with a *text conversion* function through composition or specialization (e.g., by registering an evaluable block or overriding a method).²

The semantic corpus provides methods for searching objects that are *similar* to a reference object or to a natural-language *query* string. For each search, it answers a `SearchResultSet`, which contains a *relevance score* for each object in the corpus.³ A result set can be *ranked* into a `SearchResultRanking` of a finite length using the different ranking methods discussed in section 5.3, such as top-k selection and probabilistic sampling. Additionally, the semantic corpus also supports cluster-based selection of representative objects without a reference object.

²Currently accessible through `SemanticPluggableCorpus`.

³At the time of writing, these classes have not yet been extracted from the `SemanticCorpus` in our public implementation.

Example. A programmer wants to find classes in the system that implement means for semantic search. For this, they can create a semantic corpus of all classes based on their names and comments, perform a search, and rank the results:

```
corpus := self systemNavigation allClasses
  asSemanticCorpusWithTitle: #name
  content: #comment.
results := corpus findObjectsForQuery: 'semantic
search database'.
ranking := results top: 5.
ranking →a a SearchResultRanking(
#SemanticCorpus->0.533
#SemanticHelpSearchTopic->0.442
#SemanticText->0.385 #SemanticAgentParser->0.364
#SemanticMathAgent->0.338)
```

^aWe use the notation `<expr> → <result>` to indicate a *print-it* evaluation [67, p. 13].

Conversations

To support text generation and machine reasoning, we integrate different conversational LLMs into SEMANTICTEXT. To this end, the framework defines a `SemanticConversation` class (fig. A.2). A conversation consists of a sequence of `Messages`, each of which is specified with a `role` (either `system`, `user`, or `assistant`) and a `text`. A programmer can set up a conversation with a sequence of initial messages, such as a `system` message for providing instructions and a `user` message for the first question of the user, and ask the conversation to complete itself, which will request the (stateless) LLM and append a new generated `assistant` message to the conversation.

To allow for the construction of agents, the conversation can also be equipped with a `FunctionSpec` that provides one or multiple `Functions`. Each function has a `name`, can specify an optional list of arguments and their constraints as well as a `description`, and contains an `action` object (e.g., a block closure). When the LLM is invoked, it can issue one or multiple `FunctionCalls` based on the function specification as part of the generated `assistant` message. The conversation automatically resolves these function calls by calling each issued function with the generated arguments, storing the result in a new `FunctionMessage`, and providing all function messages to the LLM in another request, which uses the results for generating the next `assistant` message.

Analogously to semantic corpora, each conversation is configured through a `ConversationModel` reference and a `ConversationConfig` for controlling the

A. Accessing Semantic Technologies through the SEMANTICTEXT Framework

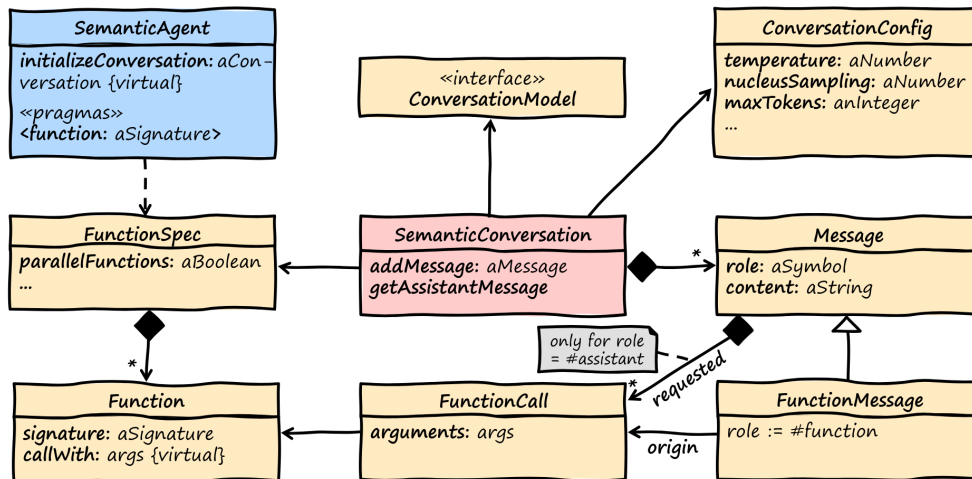


Figure A.2.: The general object model for conversational agents in SEMANTICTEXT (UML class diagram with same semantics as before). Programmers can create a `SemanticConversation` (red) with a set of messages and request an answer from a conversation model or implement an Agent class (blue) and declare callable functions via pragmas on it.

generation behavior through parameters such as the model temperature and the maximum number of tokens to generate.

Example. A programmer wants to create a chatbot that can retrieve the current time and date. For this, they define a conversation with an appropriate configuration for the LLM, define the necessary functions, and provide the question of the user:

```

SemanticConversation new
  withConfigDo: [:config |
    config temperature: 0.2];
  addFunction: #getTime action: [Time now];
  addUserMessage: 'What time is it?';
  getAssistantReply → 'The current time is
13:59.'

```

Additionally, SEMANTICTEXT supports the construction of class-based agents through the base class `SemanticAgent`, which clients can specialize to provide custom instructions and functions. It provides a custom DSL (domain-specific language) for specifying function signatures with its arguments, their constraints (such as argument types and optional arguments), and their descriptions through method pragmas. The agent class will automatically dispatch function calls and handle errors from called functions.

Example. A programmer wants to build a chatbot that can access the running Squeak image to assist the user. To achieve this, they create a subclass of `SemanticAgent`, initialize the conversation, and define an `#eval:` method:

```
SemanticAgent subclass: #SemanticSqueakAgent
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'SemanticText-Model-Agents'

SemanticSqueakAgent>initializeConversation:
aConversation
  super initializeConversation: aConversation.
  aConversation addSystemMessage: 'You are a
  Squeak/Smalltalk assistant.'.

SemanticSqueakAgent>eval: aString
  "Evaluate a Smalltalk expression in the running
  Squeak image."
  <function: eval(
    expression: string "e.g. '(8 nthRoot: 3)-1'"
  )>
  ^ Compiler evaluate: aString
```

Finally, the programmer invokes the agent:

```
SemanticSqueakAgent makeNewConversation
  addUserMessage: 'how many windows are open';
  getAssistantReply → 'You currently have 138
  open windows in your Squeak environment.'
```

A.2. Language Providers

SEMANTICTEXT defines interfaces for text embedding and generation models that can be implemented by different *providers*. As the default provider, we implemented a basic client for the OpenAI API.⁴ The client accesses the HTTP endpoints using Squeak's `WebClient` package and supports token-based authentication and rate limits.

⁴*OpenAI API*. URL: <http://archive.today/2024.06.30-211140/https://platform.openai.com/docs/overview>

A. Accessing Semantic Technologies through the SEMANTICTEXT Framework

To support receiving generated text chunks incrementally and avoid delays, the client can handle *server-sent events*⁵ emitted by the API, transform them into an internal Generator, and update the returned SemanticMessage asynchronously for each new chunk. Clients that request streamed completions initially receive an empty SemanticMessage and can subscribe to observer signals for incorporating each appended chunk into their data model or interface (such as a graphical display).

Next to the OpenAI provider, SEMANTICTEXT also implements a *mocking provider* for simulating real language models (see section A.3).

A.3. Tool Support

Developing semantic applications (i.e., programs that rely on semantic technologies) presents particular challenges to programmers, including nondeterministic behavior, latencies, and cost of language models.

First, language models behave *nondeterministically* in many settings, which can impede testing. In addition, generative LLMs exhibit characteristics of a chaotic system, meaning that small changes in the provided input might cause large changes in their outputs. For example, if the user phrases the same question using slightly different words, an agent might interpret the task very differently and fail to solve it appropriately. Thus, the behavior of LLMs is often poorly predictable.

Second, requesting language models to process large amounts of information can involve notable *latencies*, which result from the intrinsic speed of models and possible rate limits of APIs. For example, embedding or describing all methods in a typical Squeak image using contemporary OpenAI models might take several minutes (see section 9.2). These delays can reduce the experience of temporal immediacy for programmers who are prototyping or testing semantic applications.

Third, *pecuniary costs* often arise when using cloud models. Sending a single message to an object or evaluating a simple script during prototyping might suddenly involve expenses of several (US) cents or sometimes even dollars. These expenses are largely unexpected, invisible, and difficult to track. This can result not only in unintended invoices but also overcautiousness in developers, forming a new kind of psychological barrier to the experienced immediacy of programmers.

Through the SEMANTICTEXT framework, we provide different tools for addressing these challenges and thus improving the development experience

⁵“Server-Sent Events”. In: *HTML Living Standard*, section 9.2. WHATWG, 2024-06-26. URL: <https://html.spec.whatwg.org/commit-snapshots/25aaad7f6a10785efe041fb05a597400e700ef10/#server-sent-events>.

of semantic-application programmers: a *conversation editor* for prototyping generative agents, a *mocking provider* for replacing slow and expensive language models, and means for *price estimation* and *expense watching*.

Conversation Editor

The *conversation editor* is a graphical tool through which programmers can prototype and debug generative agents. It allows programmers to inspect and modify all parts of a conversation, such as messages, functions, and function calls, using a text-based WYSIWYG⁶ interface and a custom DSL.

For example, programmers can engage in prompt engineering by drafting different versions of system instructions and function specifications and comparing their effects (by *duplicating* the conversation window using the eponymous Morphic halo handle in Squeak). Moreover, they can simulate the behavior of certain function calls or former assistant replies without implementing them. Thus, the conversation editor allows programmers to test agents and explore the efficacy of different prompts.⁷

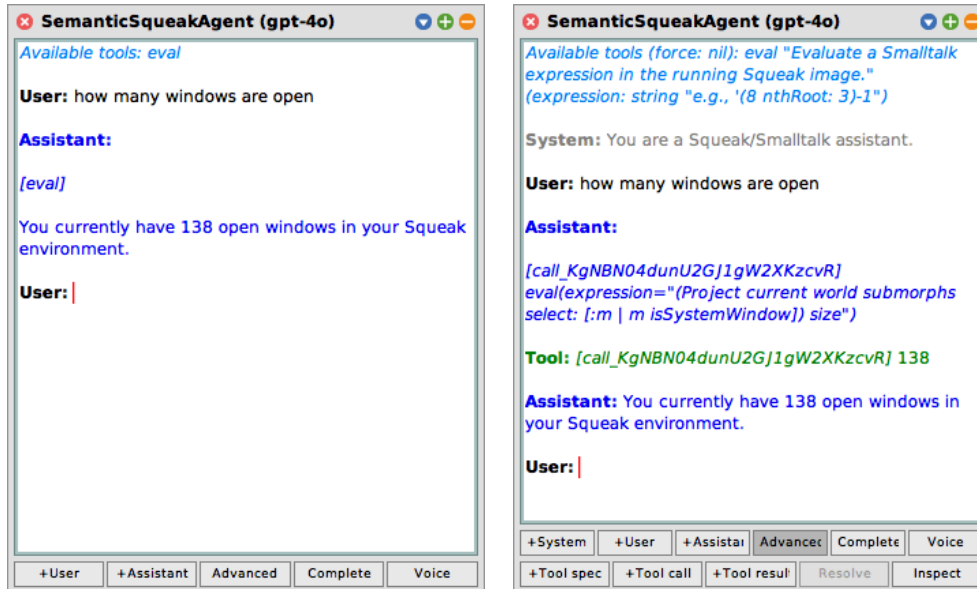
The GUI of the conversation editor provides two modes (fig. A.3): the *default mode* only shows user and assistant messages, whereas the *advanced mode* also provides read and write access to system messages as well as function specifications, function calls, and function results. So, while the advanced mode is intended for prototyping and debugging, the former mode functions as an interface for end users. The conversation editor also constitutes a composable UI component that can be reused by semantic applications such as our conversational prototype for the semantic workspace (see section 7.3).

Mocking Provider

SEMANTICTEXT implements a mocking provider with different *mocking models*, which simulate the behavior of real language models through simple heuristics. The `MockingConversationModel` answers a static assistant message for each request (which can be adjusted through parameters in the `SemanticConversationConfig`). The `MockingEmbeddingModel` embeds each document as a small vector based on the term frequencies of a short hard-coded list of words.

⁶WYSIWYG: What you see is what you get.

⁷Similar possibilities are offered by several other tools such as the *OpenAI chat playground* (<https://platform.openai.com/playground/chat>). However, existing tools are typically not integrated into the live programming system, which impedes prototyping, since programmers need to connect language models with other artifacts and logic. In addition, our conversation editor can also be used for post-mortem debugging and exploring actual conversations from a semantic application.



- (a) In the *default mode*, end users can engage in conversations with the assistant.
- (b) In the *advanced mode*, developers can declare prompts and functions and inspect or simulate function calls of the model.

Figure A.3.: The *conversation editor* for accessing and manipulating conversations with LLM agents.

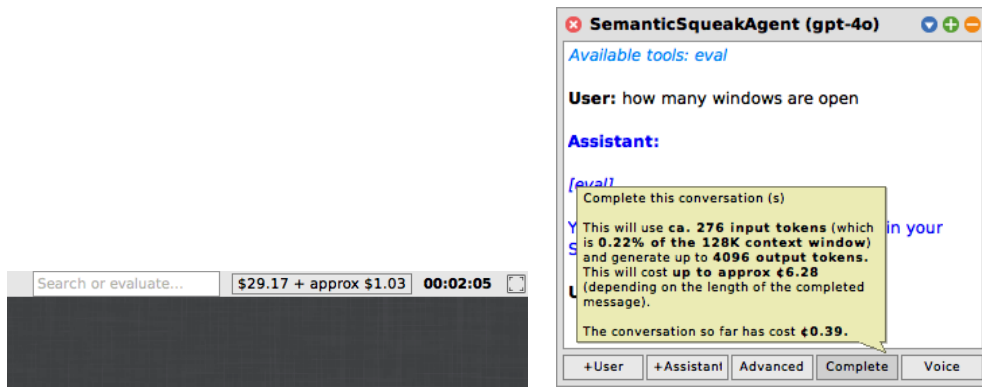
By temporarily using mocking models, programmers can ensure the deterministic behavior of semantic applications while testing them or developing other parts of a system. Finally, they can avoid additional latencies or costs.

Price Estimation and Expense Watching

To facilitate planning of expenses, model providers offer means for *estimating* the prices of requests before actually submitting them to an API. Additionally, they *log* the expenses from submitted requests. For this, they use a combination of usage statistics returned from API responses (such as the number of processed tokens) and local (approximating) cost models.

This allows for the construction of *expense watchers*, which display the expenses for semantic operations in single tools such as the conversation editor or in the global system (fig. A.4).⁸ Thus, programmers gain improved trust and transparency regarding pecuniary costs of language models.

⁸This resembles common usage dashboards of cloud services such as OpenAI (<https://platform.openai.com/usage>). However, expense watchers in SEMANTICTEXT detail costs with a finer granularity and improve visibility through direct integration into the programming system.



(a) A global expense watcher attached to the world's main docking bar displays all expenses in the system.

(b) A tooltip in the conversation editor provides detailed information on the prior costs of a conversation and estimates the tokens and price for answering the next user message.

Figure A.4.: Through *expense watchers*, programmers can track the costs of semantic expenses within the programming system.

Appendix Summary

We have described the SEMANTICTEXT framework for accessing semantic technologies in Squeak. SEMANTICTEXT models a *semantic corpus* that can emit (ranked) *search results* for embedding searches and a *conversation* with *messages* and *functions* to implement conversational *agents*. Searches and conversations can be powered using different *language model providers* such as the built-in OpenAI API client. To facilitate prototyping of semantic applications despite the nondeterminism, latencies, and cost of LLMs, SEMANTICTEXT offers different tools for editing conversations, mocking LLMs, and estimating or tracking expenses.

B. Example Conversation Log of the Exploratory Programming Agent

Here, we provide a full conversation log with our prototype of the exploratory programming agent based on the example from [fig. 8.5](#), including all (unoptimized) system prompts and function calls.

The conversation log has been generated by the `SemanticConversationEditor` of `SEMANTICTEXT` (see [section A.3](#)). For better readability, we have inserted additional spaces in the tool messages of the system and adjusted colors.

Available tools:

- `browsePackage` "get all classes in the package" (`packageName`: string)
- `browseClass` "get the definition, comment, and all method names ('selectors') of the class" (`className`: string)
- `browseMethod` "get the full source code of the method `className»selector`" (`className`: string "e.g., 'Object' or 'Object class' (for static class-side methods)", `selector`: string "e.g., 'copy', 'at:put:', or '='")
- `eval` "evaluate a Smalltalk expression (aka 'do it'). Note that all temporary variables from the expression are local to this evaluation and cannot be accessed again in later expressions." (`expression`: string)
- `browseSenders` "get a list of methods in the system that use this selector (i.e., call a method with that name). useful to find examples that teach you how you can invoke a method." (`selector`: string, `query?`: string "Optional brief description in a few words in natural language of what the selector should be used for. Useful to rank the results based on the relevance for a given context.")

System: You are a helpful assistant for exploratory programming in a Squeak/Smalltalk image. You assist the user to solve problems by searching existing code and writing new code. When answering questions about or working with existing frameworks, ALWAYS do extensive research in the system by browsing relevant classes, methods, and their senders. Depending on the complexity of the task and uncertainty of the possible outcome, "extensive" means that you should always do a broad source, pursue multiple alternatives, and browse about 3x up to 5x as many classes/methods/senders as actually might be relevant in the final conclusion. When asked coding questions, include an example

B. Example Conversation Log of the Exploratory Programming Agent

in your reply (that you have run before). DO NOT tell the user to run the code manually, but DO always and eagerly run the code by yourself (using eval())!

You have access to a couple of different functions to explore the Squeak image and try things out:

- To get an overview of the capabilities of a class, browse its protocol (using `browseClass()`).
- Sometimes, you may also need to enumerate all the classes in a package (using `browsePackage()`).
- To find out how a method selector can be used (e.g., how the provided arguments should look like), explore some of its references (using `browseSenders()`).
- To get the full signature of a method of a method (including all argument names) and its comment (if available), read its source code (using `browseMethod()`).
- Analogously, to find out how a method is implemented, read its source code as well (using `browseMethod()`).
- You can also run Smalltalk code (using `eval()`) to test things out (e.g., to look at objects, acquire information, or test whether your code works). IMPORTANT: If you write any new Smalltalk code, you MUST ALWAYS run it by yourself (using `eval()`) to make sure it is functioning BEFORE sending the code to the user. This also applies if the user asks a question like "how can i do x?".

DO NOT browse the same thing twice since it won't get better but can be expensive.

If you have trouble getting your code to work, try the following:

- Browse methods on the error stack to understand how they work and have to be used.
- Browse senders of the methods you want to use to understand how they are usually used in other contexts.

You can perform up to 25 browse/eval requests per message. Generously use that quota if the task is not trivial.

System: You are now an object (a Text) being explored by the user inside an object inspector, ready to answer questions about yourself (``self``).

To solve the task, you are encouraged to browse code (using the browse functions) and run your own code proactively (using the ``eval()`` function). You may run ``eval()`` many times until you have found a solution. When calling ``eval()``, refer to the object as ``self``. To retrieve an instance variable in Smalltalk code, use ``(self instVarNamed: 'variableName')``. If the question is of programmatic nature, it is usually best to fully translate the problem into a working code expression for the

B. Example Conversation Log of the Exploratory Programming Agent

`eval()` function and return the result of evaluating that expression. Critically reflect whether the solution makes sense before you decide that you are done.

Keep your answers as short as possible unless instructed otherwise. When possible, just answer a single word, number, or print string only. Instead of "the result is X", just say "X". DO NOT write prosa like "there are several ways" but just like the ways briefly. If the user asks longer or more open questions, you can provide more detailed answers. Provide code in your reply if the user asks for it (e.g., when they ask "how can i do X", print out a tested code snippet to the chat).

Assistant:

```
[call_fc404ee2-9c96-4980-8160-c3cfa47fb9f9] eval(expression="self")
[call_7b41cd9e-7d54-4d04-9451-fbee10fbdcc3] eval(expression="self class")
[call_1c0f6794-97de-42b8-b9ec-27634c4e3054] eval(expression="self
longPrintStringLimitedTo: 100")
[call_c704c70d-70fb-4dab-a45f-2ba446634556] browseClass(className="Text")
```

Tool: [call_fc404ee2-9c96-4980-8160-c3cfa47fb9f9] {"result": "a Text for 'ABCDE'"}

Tool: [call_7b41cd9e-7d54-4d04-9451-fbee10fbdcc3] {"result": "Text"}

Tool: [call_1c0f6794-97de-42b8-b9ec-27634c4e3054] {"result": ""string:
\\t"ABCDE"\\r\\nruns: \\ta RunArray runs: #(3 1 1) values: {{a TextEmphasis code: 1}.
#() . {a TextEmphasis code: 2}}...etc...\\r""}

Tool: [call_c704c70d-70fb-4dab-a45f-2ba446634556] {"className": "Text",
"definition": "ArrayedCollection subclass: #Text\\n\\tinstanceVariableNames: 'string
runs\\n\\tclassVariableNames: 'IgnoreStyleIfOnlyBold\\n\\tpoolDictionaries:
'TextConstants\\n\\tcategory: 'Collections-Text'", "comment": "I represent a character
string that has been marked with abstract changes in character appearance.
Actual display is performed in the presence of a TextStyle which indicates, for each
abstract code, an actual font to be used. A Text associates a set of TextAttributes
with each character in its character string. These attributes may be font numbers,
emphases such as bold or italic, or hyperling actions. Font numbers are interpreted
relative to whatever textStyle appears, along with the text, in a Paragraph. Since
most characters have the same attributes as their neighbors, the attributes are
stored in a RunArray for efficiency. Each of my instances has\\n\\tstring\\t\\ta String\\n
\\truns\\t\\ta RunArray", "categorizedSelectors": {"accessing": ["#append:",
"#applyLanguageInformation:", "#at:", "#at:put:", "#findString:startingAt:",
"#findString:startingAt:caseSensitive:", "#lineCorrespondingToIndex:",
"#lineCount", "#lineIndicesDo:", "#lineNumber:", "#lines", "#linesDo:", "#prepend:",

B. Example Conversation Log of the Exploratory Programming Agent

"comment": "I am an abstract superclass for collections that have a well-defined order associated with their elements. Thus each element is externally-named by integers referred to as indices.", "categorizedSelectors": {"accessing": ["#after:", "#after:ifAbsent:", "#allButFirst", "#allButFirst:", "#allButLast", "#allButLast:", "#any:", "#anyOne", "#at:ifAbsent:", "#at:ifPresent:", "#at:ifPresent:ifAbsent:", "#at:incrementBy:", "#atAll:", "#atAll:put:", "#atAll:putAll:", "#atAllPut:", "#atLast:", "#atLast:ifAbsent:", "#atLast:ifPresent:", "#atLast:ifPresent:ifAbsent:", "#atLast:put:", "#atPin:", "#atRandom:", "#atWrap:", "#atWrap:put:", "#before:", "#before:ifAbsent:", "#eighth", "#fifth", "#first", "#first:", "#fourth", "#from:to:put:", "#identityIndexOf:", "#identityIndexOf:ifAbsent:", "#identityIndexOf:startingAt:", "#identityIndexOf:startingAt:ifAbsent:", "#indexOf:", "#indexOf:ifAbsent:", "#indexOf:startingAt:", "#indexOf:startingAt:ifAbsent:", "#indexOfAnyOf:", "#indexOfAnyOf:ifAbsent:", "#indexOfAnyOf:startingAt:", "#indexOfAnyOf:startingAt:ifAbsent:", "#indexOfSubCollection:", "#indexOfSubCollection:startingAt:", "#indexOfSubCollection:startingAt:ifAbsent:", "#indicesOfSubCollection:", "#indicesOfSubCollection:startingAt:", "#integerAt:", "#integerAt:put:", "#last", "#last:", "#lastIndexOf:", "#lastIndexOf:ifAbsent:", "#lastIndexOf:startingAt:", "#lastIndexOf:startingAt:ifAbsent:", "#lastIndexOfAnyOf:", "#lastIndexOfAnyOf:ifAbsent:", "#lastIndexOfAnyOf:startingAt:", "#lastIndexOfAnyOf:startingAt:ifAbsent:", "#middle", "#ninth", "#replaceAll:with:", "#replaceFrom:to:with:", "#replaceFrom:to:with:startingAt:", "#second", "#seventh", "#sixth", "#size", "#swap:with:", "#third"], "adding": ["#addAllFirstTo:"], "comparing": ["#=", "#hasEqualElements:", "#hash"], "converting": ["#@", "#asArray", "#asByteArray", "#asColorArray", "#asFloat32Array", "#asFloat64Array", "#asFloatArray", "#asIntegerArray", "#asStringWithCr", "#asWordArray", "#concatenation", "#flatten", "#flattened", "#join", "#joinSeparatedBy:", "#readStream", "#readStreamWithProgress", "#reverse", "#reverseInPlace", "#reversed", "#subsequences:", "#writeStream"], "copying": ["#", "#copyAfter:", "#copyAfterLast:", "#copyEmpty", "#copyFrom:to:", "#copyLast:", "#copyReplaceAll:with:", "#copyReplaceFrom:to:with:", "#copyUpThrough:", "#copyUpTo:", "#copyUpToLast:", "#copyWith:", "#copyWithFirst:", "#copyWithoutIndex:", "#forceTo:paddingStartWith:", "#forceTo:paddingWith:", "#grownBy:"], "enumerating": ["#allButFirstDo:", "#allButLastDo:", "#asDigitsToPower:do:", "#collect:", "#collect:from:to:", "#combinations:atATimeDo:", "#do:", "#do:separatedBy:", "#do:without:", "#findBinary:", "#findBinary:do:ifNone:", "#findBinary:ifNone:", "#findBinaryIndex:", "#findBinaryIndex:do:ifNone:", "#findBinaryIndex:ifNone:", "#findFirst:", "#findFirst:startingAt:", "#findLast:", "#findLast:startingAt:", "#findNearbyBinaryIndex:", "#from:to:do:", "#groupsCollect:", "#groupsDo:", "#groupsOf:atATimeCollect:", "#groupsOf:atATimeDo:", "#keysAndValuesDo:", "#nextToLast", "#overlappingPairsCollect:", "#overlappingPairsDo:"],

B. Example Conversation Log of the Exploratory Programming Agent

```
"#overlappingPairsWithIndexDo:", "#paddedWith:do:", "#pairsCollect:", "#pairsDo:",  
"#permutationsDo:", "#piecesCutWhere:", "#piecesCutWhere:do:",  
"#polynomialEval:", "#replace:", "#reverseDo:", "#reverseWith:do:", "#select:",  
"#sesqlIndexedVariables", "#splitBy:", "#splitBy:do:", "#upTo:", "#with:collect:",  
"#with:do:", "#withIndexCollect:", "#withIndexDo:", "#withIndexDo:separatedBy:",  
"filter streaming": ["#putOn:", "printing": ["#joinOn:", "#joinOn:separatedBy:"],  
"private": ["#asDigitsAt:in:do:", "#combinationsAt:in:after:do:",  
"#copyReplaceAll:with:asTokens:", "#errorFirstObject:", "#errorLastObject:",  
"#errorOutOfBounds", "#permutationsStartingAt:do:"], "removing":  
["#remove:ifAbsent:"], "shuffling": ["#shuffle", "#shuffleBy:", "#shuffled",  
"#shuffledBy:"], "testing": ["#beginsWith:", "#beginsWithAnyOf:", "#endsWith:",  
"#endsWithAnyOf:", "#includes:", "#isSequenceable"]], "superclass":  
{ "className": "Collection", "definition": "Object subclass: #Collection\n  
\n\tinstanceVariableNames: "\n\tclassVariableNames: "\n\tpoolDictionaries: "\n  
\n\tcategory: 'Collections-Abstract'", "comment": "I am the abstract superclass of all  
classes that represent a group of elements.", "categorizedSelectors": {"accessing":  
["#any:", "#any:as:", "#anyOne", "#atRandom", "#atRandom:", "#capacity",  
"#removeAtRandom", "#size", "#skip:", "#take:"], "adapting":  
["#adaptToCollection:andSend:", "#adaptToComplex:andSend:",  
"#adaptToNumber:andSend:", "#adaptToPoint:andSend:",  
"#adaptToString:andSend:"], "adding": ["#add:", "#add:withOccurrences:",  
"#addAll:", "#addAllFirstTo:", "#addIfNotPresent:"], "arithmetic": ["#*", "#**", "#+",  
"#-", "#\V", "#\V\W", "#\W", "#raisedTo:"], "as yet unclassified":  
["#browseMethodVersions", "#browseMethods", "#browseRankedMethods",  
"#closeTo:", "#random:sample:", "#randomSample:", "#sesqlIndexedVariables"],  
"comparing": ["#hash"], "converting": ["#asArray", "#asBag", "#asByteArray",  
"#asCharacterSet", "#asCollection", "#asIdentitySet", "#asOrderedCollection",  
"#asSet", "#asSortedArray", "#asSortedCollection", "#asSortedCollection:",  
"#histogramOf:", "#topologicallySortedUsing:"], "copying": ["#", "#copyWith:",  
"#copyWithDependent:", "#copyWithout:", "#copyWithoutAll:",  
"#copyWithoutDuplicates", "#withoutDuplicates"], "enumerating": ["#allSatisfy:",  
"#anySatisfy:", "#associationsDo:", "#collect:", "#collect:as:", "#collect:into:",  
"#collect:thenDo:", "#collect:thenSelect:", "#concurrentCollect:", "#count:",  
"#detect:", "#detect:ifFound:", "#detect:ifFound:ifNone:", "#detect:ifNone:",  
"#detectMax:", "#detectMin:", "#detectSum:", "#do:", "#do:displayingProgress:",  
"#do:displayingProgress:every:", "#do:separatedBy:", "#do:without:",  
"#findFirstInByteString:startingAt:", "#fold:", "#gather:", "#groupBy:",  
"#groupBy:having:", "#inject:into:", "#noneSatisfy:", "#occurrencesOf:",  
"#orderedGroupBy:", "#orderedGroupBy:having:", "#reduce:", "#reject:",  
"#reject:thenDo:", "#select:", "#select:thenCollect:", "#select:thenDo:", "#try:",  
"#try:ignore:", "#try:ignore:ifException:", "#try:ignore:logged:"], "filter streaming":  
["#contents", "#flattenOnStream:", "#write:"], "math functions": ["#abs", "#arcCos",
```

B. Example Conversation Log of the Exploratory Programming Agent

```
"#arcSin", "#arcTan", "#average", "#ceiling", "#cos", "#degreeCos", "#degreeSin",
"#exp", "#floor", "#ln", "#log", "#log2", "#max", "#min", "#minMax", "#negated",
"#range", "#reciprocal", "#roundTo:", "#rounded", "#sign", "#sin", "#sqrt",
"#squared", "#sum", "#tan", "#truncated"], "printing": ["#name",
"#printElementsOn:", "#printElementsOn:separatedBy:", "#printNameOn:",
"#printOn:", "#storeOn:"], "printing - obsolete": ["#asCommaString",
"#asCommaStringAnd", "#asStringOn:delimiter:", "#asStringOn:delimiter:last:",
"#printOn:delimiter:", "#printOn:delimiter:last:"], "private": ["#emptyCheck",
"#errorDifferentSize", "#errorEmptyCollection", "#errorKeyNotFound:",
"#errorNoMatch", "#errorNotFound:", "#errorNotKeyed", "#fillFrom:with:",
"#isOfSizeCheck:"], "removing": ["#remove:", "#remove:ifAbsent:",
"#removeAll", "#removeAll:", "#removeAllFoundIn:", "#removeAllSuchThat:"], "set
logic": ["#difference:", "#intersection:", "#symmetricDifference:", "#union:"],
"sorting": ["#sorted", "#sorted:", "#sortedSafely"], "testing": ["#contains:",
"#identityIncludes:", "#ifEmpty:", "#ifEmpty:ifNotEmpty:", "#ifEmpty:ifNotEmptyDo:",
"#ifNotEmpty:", "#ifNotEmpty:ifEmpty:", "#ifNotEmptyDo:",
"#ifNotEmptyDo:ifEmpty:", "#includes:", "#includesAllOf:", "#includesAnyOf:",
"#includesSubstringAnywhere:", "#isCollection", "#isEmpty", "#isEmptyOrNil",
"#isSequenceable", "#isSignedIntegerArray", "#isUnsignedIntegerArray", "#isZero",
"#notEmpty"], "superclass": {"className": "Object"}, "classSide": {"className":
"Collection class", "definition": "Collection class\n\tinstanceVariableNames: """,
"ategorizedSelectors": {"class initialization": ["#initialize"], "instance creation":
["#ofSize:", "#with:", "#with:with:", "#with:with:with:", "#with:with:with:with:",
"#with:with:with:with:with:", "#with:with:with:with:with:with:", "#withAll:"], "testing":
["#isAbstract"]}}, "classSide": {"className": "SequenceableCollection class",
"definition": "SequenceableCollection class\n\tinstanceVariableNames: """,
"ategorizedSelectors": {"stream creation": ["#new:streamContents:",
"#streamContents:", "#streamContents:limitedTo:"], "testing": ["#isAbstract"]}},
"classSide": {"className": "ArrayedCollection class", "definition":
"ArrayedCollection class\n\tinstanceVariableNames: """, "ategorizedSelectors":
{"instance creation": ["#new", "#new:withAll:", "#newFrom:", "#newFromStream:",
"#with:", "#with:with:", "#with:with:with:", "#with:with:with:with:",
"#with:with:with:with:with:", "#with:with:with:with:with:with:", "#withAll:"], "testing":
["#isAbstract"]}}, "classSide": {"className": "Text class", "definition": "Text class\n
\tinstanceVariableNames: """, "ategorizedSelectors": {"class initialization":
["#initTextConstants", "#initialize", "#unload"], "file services":
["#fileReaderServicesForFile:suffix:", "#openFromHtmlStream:",
"#serviceOpenHtmlFile", "#services"], "filler text": ["#THEQUICKBROWNFOX",
"#allDigits", "#codeSample", "#forssmanSample", "#hamburgetonstiv",
"#loremIpsum", "#melvilleSample", "#symbolSample", "#textSample",
"#theQuickBrownFox"], "instance creation": ["#empty", "#fromString:", "#new:",
"#new:streamContents:", "#streamContents:", "#string:attribute:"],
```

B. Example Conversation Log of the Exploratory Programming Agent

```
"#string:attributes:", "#string:emphasis:"], "preferences": [{"#ignoreStyleIfOnlyBold", "#ignoreStyleIfOnlyBold:"}, {"private": [{"#addAttribute:toArray:", "#string:runs:"}]}}}
```

Assistant: How can I help you?

User: what attributes are in this text

Assistant:

```
[call_8A3JIAZ0NVQNHpQJRXSk9k5y] eval(expression="self runs collect: [:each | each attributes]")
```

Tool: [call_8A3JIAZ0NVQNHpQJRXSk9k5y] {"error": {"description": "MessageNotUnderstood: Array»attributes", "stack": ["Array(Object)»doesNotUnderstand: #attributes", "[] in Text»Dolt", "RunArray(SequenceableCollection)»collect:", "Text»Dolt", "Compiler»evaluateCue:ifFail:logged:", "Compiler»evaluateCue:ifFail:", "Compiler»evaluate:in:to:notifying:ifFail:", "Compiler class»evaluate:for:notifying:", "Compiler class»evaluate:for:"], "note": "A common source of MessageNotUnderstood errors is that some brackets were missed to separate messages from each other. E.g., `myArray select: [:x | x notNil] collect: [:x | x customer]` must be corrected to `(myArray select: [:x | x notNil]) collect: [:x | x customer]`, and `myArray select: [:x | x notNil] , otherArray` must be corrected to `(myArray select: [:x | x notNil]) , otherArray`."}

Assistant:

```
[call_Jn24LniveyZexVuGH1xwYpJm] eval(expression="self runs collect: [:each | each values collect: [:value | value printString]]")
```

Tool: [call_Jn24LniveyZexVuGH1xwYpJm] {"error": {"description": "MessageNotUnderstood: Array»values", "stack": ["Array(Object)»doesNotUnderstand: #values", "[] in Text»Dolt", "RunArray(SequenceableCollection)»collect:", "Text»Dolt", "Compiler»evaluateCue:ifFail:logged:", "Compiler»evaluateCue:ifFail:", "Compiler»evaluate:in:to:notifying:ifFail:", "Compiler class»evaluate:for:notifying:", "Compiler class»evaluate:for:"], "note": "A common source of MessageNotUnderstood errors is that some brackets were missed to separate messages from each other. E.g., `myArray select: [:x | x notNil] collect: [:x | x customer]` must be corrected to `(myArray select: [:x | x notNil]) collect: [:x | x customer]`, and `myArray select: [:x | x notNil] , otherArray` must be corrected to `(myArray select: [:x | x notNil]) , otherArray`."}

B. Example Conversation Log of the Exploratory Programming Agent

the Hi There appears in the paragraph. If selection has no angle brackets, use the whole thing as both the text and the action.

`\n\nTextDolt – eval as a Smalltalk expression (the part in angle brackets)`

`\n\nTextLink – Show a method, class comment, class hierarchy, or class definition.`

`\n\t<Point extent:>, <Point Comment>, <Point Hierarchy>, or <Point Definition> are what you type.`

`\n\nTextURL – Show the web page. <www.disney.com>`

These attributes of text need to be stored on the disk in a regular file-out. It is done in this form: `\tHi There \n\n` in the text, and a Run containing `dSmalltalk beep;;\n\tClick here to see the extent: \n\n` in the text, and a Run containing method `LPoint extent;\nSee RunArray class scanFrom: where decoding is done.\n`, "categorizedSelectors": {"accessing": ["#anchoredMorph", "#emphasisCode", "#emphasizeScanner:", "#menu", "#menuFor:", "#oldEmphasisCode:", "#set"], "fileIn/fileOut": ["#writeScanOn:"], "html": ["#closeHtmlOn:", "#openHtmlOn:", "#shouldFormBlocks"], "initialize-release": ["#reset", "#turnOff"], "mouse events": ["#actOnClickFor:", "#actOnClickFor.in:", "#actOnClickFor.in:at:", "#actOnClickFor.in:at:editor:"], "private": ["#forFontInStyle:do:"], "testing": ["#couldDeriveFromPrettyPrinting", "#dominatedByCmd0", "#dominates:", "#isKern", "#isOblivious", "#isTextAlignment", "#isTextFontChange", "#isTextIndent", "#mayActOnClick", "#maybeExtended"]}, "superclass": {"className": "Object"}, "classSide": {"className": "TextAttribute class", "definition": "TextAttribute class\n\n\tinstanceVariableNames: \"\", \"categorizedSelectors\": {\"fileIn/Out\": [\"#classFor:\", \"#consumeIdentifierFrom:\", \"#scanCharacter\", \"#scanCharacters\", \"#scanFrom:\"], \"instance creation\": [\"#newFrom:\"]}}, \"classSide\": {\"className\": \"TextEmphasis class\", \"definition\": \"TextEmphasis class\n\n\tinstanceVariableNames: \"\", \"categorizedSelectors\": {\"fileIn/Out\": [\"#consumeIdentifierFrom:\", \"#scanCharacters\", \"#scanFrom:\"], \"instance creation\": [\"#bold\", \"#italic\", \"#narrow\", \"#normal\", \"#struckOut\", \"#underlined\"]}}

Assistant: The codes for TextEmphasis mean the following:

- 1: bold
- 2: italic
- 4: underlined
- 8: narrow
- 16: struck out

User: multiple different ways to make self all italic? answer only code!

Assistant:

```
[call_NY8slOsAYQrCmiCgRD89Pvk7] eval(expression="self asText addAttribute:
TextEmphasis italic")
```


B. Example Conversation Log of the Exploratory Programming Agent

[call_5G9LNqbEmziGhP36kJXhHMRI] eval(expression="self addAttribute: TextEmphasis italic from: 1 to: self size")

[call_J3VFBGnQaDUU8FaFAtdIS4W] eval(expression="Text string: self string attribute: TextEmphasis italic")

Tool: [call_NY8sIOsAYQrCmiCgRD89Pvk7] {"result": "a Text for 'ABCDE'"}
(Note: The text "a Text for 'ABCDE'" is italicized in the original image)

Tool: [call_5G9LNqbEmziGhP36kJXhHMRI] {"result": "a Text for 'ABCDE'"}
(Note: The text "a Text for 'ABCDE'" is italicized in the original image)

Tool: [call_J3VFBGnQaDUU8FaFAtdIS4W] {"result": "a Text for 'ABCDE'"}
(Note: The text "a Text for 'ABCDE'" is italicized in the original image)

Assistant:

- self addAttribute: TextEmphasis italic
- self addAttribute: TextEmphasis italic from: 1 to: self size
- Text string: self string attribute: TextEmphasis italic

C. Recommendations for Tool Developers

Here, we provide practical recommendations for tool developers who consider integrating semantic technologies or AI into programming systems.

Remark. Note that this collection is based solely on our subjective experiences and challenges from this work. It does not aim to be exhaustive, align with best practices, or rely on significant evidence from other work.

1. **Consider the limited accuracy of semantic technologies.** Before designing a new tool or interaction, prototype critical invocations of language models in a playground and approximate a minimum viable configuration there (such as input data, prompts, or hyperparameters).
Make sure to evaluate whether model responses are sufficiently reasonable, correct, and useful for the intended applications. Communicate the risk of hallucinations to programmers.
2. **Design means for bidirectional cooperation.** Share comprehensive context and artifacts of programmers with semantic technologies to enable conceptual support by the system. Allow programmers to intervene in the work of semantic agents by letting them inspect their internal steps (such as inner monologue, function calling, and selected context), provide feedback or request changes, and choose between multiple options.
Explaining the behavior of large language models at different levels may be challenging, but it can benefit both programmers (for validating responses and building up on them) and tool developers (for evaluating and debugging semantic applications).
3. **Display the progress and cost of semantic operations.** Show progress bars or stream responses from LLMs continuously to reduce visible latencies. Provide easy access to monitor or predict the financial cost of semantic tools, or implement rate limits to avoid fear of unforeseen expenses.
4. **Optimize semantic applications.** Favor smaller, more efficient, and open-source language models when possible. Systematically tune LLM prompts and preprocess documents before embedding them to improve structure and reduce size. Consider fine-tuning language models for specific use cases.

C. Recommendations for Tool Developers

5. **Collect data for training and evaluation early on.** Logging all requests and responses from language models or semantic tools even in an early stage of prototypes can create a valuable asset for discussing prompt-tuning decisions, avoiding model drift, or fine-tuning models later.
6. **Pay attention to ethical and legal concerns.** Bear in mind disadvantages of large language models such as unsafe biases, high resource consumption, and unfair practices along their supply chain, and favor responsible and sustainable options. Respect applicable laws and liabilities such as privacy, intellectual property, and accountability by requiring programmers to opt in for semantic features, informing them about data collection, transfer, and usage and their rights, and anonymizing any collected data.

In the practice of many programming tools, tool developers can avoid most legal responsibilities by requiring programmers to build or run applications from source and bring their own API keys.

7. **Consider traditional implementations.** While ideating and prototyping new tools or features, see semantic technologies as a placeholder for any type of implementation. Discuss whether required functionalities can be best served by traditional algorithms and heuristics (such as parsers or decision trees), human interactions (e.g., by requesting brief annotations from programmers), or semantic technologies (when complex reasoning, creativity, or common-sense knowledge is required).

Bibliography

Programming and Software Engineering

- [1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. 2nd edition. MIT Press, 1996. ISBN: 978-0-262-51087-5 (cited on page 31).
- [2] Abdulaziz Alaboudi and Thomas D. LaToza. “Supporting Software Engineering Research and Education by Annotating Public Videos of Developers Programming”. In: *Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering*. CHASE '19. Montreal, Quebec, Canada: IEEE, 2019, pages 117–118. DOI: [10.1109/CHASE.2019.00035](https://doi.org/10.1109/CHASE.2019.00035) (cited on page 99).
- [3] Tyler Angert, Miroslav Suzara, Jenny Han, Christopher Pondoc, and Hariharan Subramonyam. “Spellburst: A Node-Based Interface for Exploratory Creative Coding with Natural Language Prompts”. In: *Proceedings of the 36th Annual Symposium on User Interface Software and Technology*. UIST '23. San Francisco, CA, USA: Association for Computing Machinery, 2023. DOI: [10.1145/3586183.3606719](https://doi.org/10.1145/3586183.3606719) (cited on page 96).
- [4] Shraddha Barke, Michael B. James, and Nadia Polikarpova. “Grounded Copilot: How Programmers Interact with Code-Generating Models”. In: *Proceedings of the ACM on Programming Languages*. OOPSLA 7.1 (Apr. 2023). DOI: [10.1145/3586030](https://doi.org/10.1145/3586030) (cited on pages 13, 20, 96).
- [5] Kent Beck. *Extreme Programming Explained: Embrace Change*. USA: Addison-Wesley, 2000. ISBN: 978-0-201-61641-5 (cited on page 95).
- [6] Tom Beckmann, Stefan Ramson, Patrick Rein, and Robert Hirschfeld. “Visual Design for a Tree-Oriented Projectional Editor”. In: *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming*. <Programming> '20 Companion. Porto, Portugal: Association for Computing Machinery, 2020, pages 113–119. DOI: [10.1145/3397537.3397560](https://doi.org/10.1145/3397537.3397560) (cited on page 79).

- [7] Marcel Bruch, Thorsten Schäfer, and Mira Mezini. “FrUIT: IDE Support for Framework Understanding”. In: *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology EXchange*. eclipse '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pages 55–59. DOI: [10.1145/1188835.1188847](https://doi.org/10.1145/1188835.1188847) (cited on page 95).
- [8] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume*. Volume 1: A System of Patterns. Wiley, 1996. ISBN: 978-0-471-95869-7 (cited on page 39).
- [9] Cristian Cadar and Koushik Sen. “Symbolic Execution for Software Testing: Three Decades Later”. In: *Communications of the ACM* 56.2 (Feb. 2013), pages 82–90. ISSN: 0001-0782. DOI: [10.1145/2408776.2408795](https://doi.org/10.1145/2408776.2408795) (cited on page 12).
- [10] Andrei Chiş, Oscar Nierstrasz, Aliaksei Syrel, and Tudor Gîrba. “The Moldable Inspector”. In: *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2015. Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pages 44–60. DOI: [10.1145/2814228.2814234](https://doi.org/10.1145/2814228.2814234) (cited on pages 8, 10).
- [11] Dave Collins and Dave Collins. *Designing Object-Oriented User Interfaces*. Object-Oriented Software Engineering Series. Redwood City, CA: Benjamin Cummings, 1995. ISBN: 978-0-8053-5350-1 (cited on page 75).
- [12] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie Van Deursen, and Jarke J. van Wijk. “Execution Trace Analysis through Massive Sequence and Circular Bundle Views”. In: *Elsevier Journal of Systems and Software* 81.12 (2008), pages 2252–2268. ISSN: 0164-1212. DOI: [10.1016/j.jss.2008.02.068](https://doi.org/10.1016/j.jss.2008.02.068) (cited on page 12).
- [13] Mihaly Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. 1st edition. New York: Harper Perennial, July 2008. ISBN: 978-0-06-133920-2 (cited on pages 2, 14, 90).
- [14] Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers. “The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse”. In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. PPPJ '11. Kongens Lyngby, Denmark: Association for Computing Machinery, 2011, pages 71–80. DOI: [10.1145/2093157.2093168](https://doi.org/10.1145/2093157.2093168) (cited on page 96).
- [15] Damian Frolich and L. Thomas van Binsbergen. “A Generic Back-End for Exploratory Programming”. In: *Trends in Functional Programming: 22nd International Symposium, Revised Selected Papers*. Edited by Viktória Zsóok and John Hughes. TFP 2021. Virtual Event: Springer, Feb. 2021, pages 24–43. DOI: [10.1007/978-3-030-83978-9_2](https://doi.org/10.1007/978-3-030-83978-9_2) (cited on page 8).

- [16] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1984. ISBN: 978-0-201-11372-3 (cited on pages 9, 10, 12, 95).
- [17] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. USA: Addison-Wesley, 1983. ISBN: 978-0-201-11371-6 (cited on pages 1, 8, 9, 12).
- [18] Jan Hartman, Rishabh Mehrotra, Hitesh Sagtani, Dominic Cooney, Rafal Gajdulewicz, Beyang Liu, Julie Tibshirani, and Quinn Slack. “AI-Assisted Coding with Cody: Lessons from Context Retrieval and Evaluation for Code Recommendations”. In: *18th ACM Conference on Recommender Systems*. RecSys '24. To appear. Bari, Italy: Association for Computing Machinery, Oct. 2024. DOI: [10.1145/3640457.3688060](https://doi.org/10.1145/3640457.3688060). arXiv: [2408.05344](https://arxiv.org/abs/2408.05344) [cs.IR] (cited on page 97).
- [19] Michael Haupt, Michael Perscheid, and Robert Hirschfeld. “Type Harvesting: A Practical Approach to Obtaining Typing Information in Dynamic Programming Languages”. In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. SAC '11. TaiChung, Taiwan: Association for Computing Machinery, 2011, pages 1282–1289. DOI: [10.1145/1982185.1982464](https://doi.org/10.1145/1982185.1982464) (cited on pages 13, 50).
- [20] Martin Hentschel, Richard Bubel, and Reiner Hähnle. “The Symbolic Execution Debugger (SED): A Platform for Interactive Symbolic Execution, Debugging, Verification and More”. In: *International Journal on Software Tools for Technology Transfer* 21.5 (Oct. 2019), pages 485–513. ISSN: 1433-2779. DOI: [10.1007/s10009-018-0490-9](https://doi.org/10.1007/s10009-018-0490-9) (cited on page 13).
- [21] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. “Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself”. In: *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Volume 32. OOPSLA '97 10. Atlanta, Georgia, USA: Association for Computing Machinery, Oct. 1997, pages 318–326. DOI: [10.1145/263700.263754](https://doi.org/10.1145/263700.263754) (cited on page 9).
- [22] Daniel Ingalls. “The Evolution of Smalltalk: From Smalltalk-72 through Squeak”. In: *Proceedings of the ACM on Programming Languages* 4.HOPL (June 2020). DOI: [10.1145/3386335](https://doi.org/10.1145/3386335) (cited on pages 9, 68).
- [23] Mary Beth Kery and Brad A. Myers. “Exploring Exploratory Programming”. In: *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2017, pages 25–29. DOI: [10.1109/VLHCC.2017.8103446](https://doi.org/10.1109/VLHCC.2017.8103446) (cited on pages 5, 8).

- [24] Amy J. Ko and Brad A. Myers. “Designing The Whyline: A Debugging Interface for Asking Questions about Program Behavior”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’04. Vienna, Austria: Association for Computing Machinery, 2004, pages 151–158. DOI: [10.1145/985692.985712](https://doi.org/10.1145/985692.985712) (cited on pages 12, 96).
- [25] Jan-Peter Krämer, Thorsten Karrer, Jonathan Diehl, and Jan Borchers. “Stackplorer: Understanding Dynamic Program Behavior”. In: *Adjunct Proceedings of the 23rd Annual ACM Symposium on User Interface Software and Technology*. UIST ’10. New York, New York, USA: Association for Computing Machinery, 2010, pages 433–434. DOI: [10.1145/1866218.1866257](https://doi.org/10.1145/1866218.1866257) (cited on page 95).
- [26] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. “The Road to Live Programming: Insights from the Practice”. In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pages 1090–1101. DOI: [10.1145/3180155.3180200](https://doi.org/10.1145/3180155.3180200) (cited on page 88).
- [27] Kimio Kuramitsu, Yui Obara, Miyu Sato, and Momoka Obara. “KOGI: A Seamless Integration of ChatGPT into Jupyter Environments for Programming Education”. In: *Proceedings of the 2023 SIGPLAN International Symposium on SPLASH-E*. Cascais, Portugal: Association for Computing Machinery, 2023, pages 50–59. DOI: [10.1145/3622780.3623648](https://doi.org/10.1145/3622780.3623648) (cited on page 97).
- [28] Kyla Levin, Nicolas van Kempen, Emery D. Berger, and Stephen N. Freund. *ChatDBG: An AI-Powered Debugging Assistant*. 2024. arXiv: [2403.16354](https://arxiv.org/abs/2403.16354) [cs.SE] (cited on page 96).
- [29] Bil Lewis. “Debugging Backwards in Time”. In: *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*. Volume cs.SE/0310016. 2003. arXiv: [cs/0310016](https://arxiv.org/abs/cs/0310016) [cs.SE] (cited on page 12).
- [30] Adrian Lienhard, Stéphane Ducasse, and Tudor Gîrba. “Taking an Object-Centric View on Dynamic Information with Object Flow Analysis”. In: *ESUG 2007 International Conference on Dynamic Languages*. Volume 35. ESUG/ICDL 2007 1. 2009, pages 63–79. DOI: [10.1016/j.cl.2008.05.006](https://doi.org/10.1016/j.cl.2008.05.006) (cited on page 12).
- [31] Toni Mattis. *Semantic Code Models for Concept-Aware Programming Environments*. Lecture Notes. Nanjing, China: HPI Research School, Nov. 2018. URL: <https://toni.mattis.berlin/files/2018-mattis-semantic-code-models.pdf> (cited on page 82).

- [32] Sean McDirmid. “Usable Live Programming”. In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pages 53–62. DOI: [10.1145/2509578.2509585](https://doi.org/10.1145/2509578.2509585) (cited on page 8).
- [33] Kai Mei, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang. *AIOS: LLM Agent Operating System*. 2024. arXiv: [2403.16971](https://arxiv.org/abs/2403.16971) [cs.O5] (cited on page 97).
- [34] Rada Mihalcea, Hugo Liu, and Henry Lieberman. “NLP (Natural Language Processing) for NLP (Natural Language Programming)”. In: *Computational Linguistics and Intelligent Text Processing*. Edited by Alexander Gelbukh. Volume 3878. CICLing ’06. Berlin, Heidelberg: Springer, 2006, pages 319–330. DOI: [10.1007/11671299_34](https://doi.org/10.1007/11671299_34) (cited on page 97).
- [35] Emerson Murphy-Hill, Rahul Jiresal, and Gail C. Murphy. “Improving Software Developers’ Fluency by Recommending Development Environment Commands”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE ’12. Cary, North Carolina: Association for Computing Machinery, 2012. DOI: [10.1145/2393596.2393645](https://doi.org/10.1145/2393596.2393645) (cited on page 95).
- [36] Brad A. Myers, John F. Pane, and Amy J. Ko. “Natural Programming Languages and Environments”. In: *Communications of the ACM* 47.9 (Sept. 2004), pages 47–52. ISSN: 0001-0782. DOI: [10.1145/1015864.1015888](https://doi.org/10.1145/1015864.1015888) (cited on page 96).
- [37] Donald A. Norman. “Cognitive Engineering”. In: *User-Centered System Design: New Perspectives on Human-Computer Interaction*. Hillsdale, NJ, USA: Lawrence Erlbaum Associates, Jan. 1986, pages 31–61. ISBN: 978-0-367-80732-0 (cited on pages 14, 89).
- [38] Richard Pawson and Robert Matthews. “Naked Objects: A Technique for Designing More Expressive Systems”. In: *SIGPLAN Not.* 36.12 (Dec. 2001), pages 61–67. ISSN: 0362-1340. DOI: [10.1145/583960.583967](https://doi.org/10.1145/583960.583967) (cited on page 75).
- [39] Michael Perscheid and Robert Hirschfeld. “Follow the Path: Debugging Tools for Test-Driven Fault Navigation”. In: *2014 Tool Demo Track of the 1st Conference on Software Maintenance, Reengineering, and Reverse Engineering Software Evolution Week*. CSMR-WCRE ’14. Los Alamitos, CA, USA: IEEE Computer Society, Feb. 2014, pages 446–449. DOI: [10.1109/CSMR-WCRE.2014.6747215](https://doi.org/10.1109/CSMR-WCRE.2014.6747215) (cited on pages 12, 96).

- [40] Guillaume Pothier and Éric Tanter. “Back to the Future: Omniscient Debugging”. In: *IEEE Software* 26.6 (2009), pages 78–85. DOI: [10.1109/MS.2009.169](https://doi.org/10.1109/MS.2009.169) (cited on page 12).
- [41] Chen Qian, Xin Cong, Wei Liu, Cheng Yang, Weize Chen, Yusheng Su, Yufan Dang, Jiahao Li, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. *Communicative Agents for Software Development*. 2023. arXiv: [2307.07924](https://arxiv.org/abs/2307.07924) [cs.SE] (cited on page 21).
- [42] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. “Babylonian-Style Programming: Design and Implementation of an Integration of Live Examples into General-Purpose Source Code”. In: *The Art, Science, and Engineering of Programming* 3.3 (Feb. 2019). ISSN: 2473-7321. DOI: [10.22152/programming-journal.org/2019/3/9](https://doi.org/10.22152/programming-journal.org/2019/3/9) (cited on page 12).
- [43] Eric Steven Raymond. *The Art of UNIX Programming*. Boston: Addison-Wesley, 2003. ISBN: 978-0-13-142901-7 (cited on page 8).
- [44] Nico Rehwaldt. “Exploring Run-Time Behavior in Reversible Experiments”. Master’s Thesis. Hasso Plattner Institute, 2012. URL: <https://nixis.de/~nikku/uni/master/thesis/exploring-run-time-behavior-nre-2012.pdf> (cited on page 61).
- [45] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. “Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness”. In: *The Art, Science, and Engineering of Programming* 3.1 (July 2019). DOI: [10.22152/programming-journal.org/2019/3/1](https://doi.org/10.22152/programming-journal.org/2019/3/1) (cited on pages 1, 5, 8).
- [46] Patrick Rein, Marcel Taeumel, and Robert Hirschfeld. “Towards Empirical Evidence on the Comprehensibility of Natural Language Versus Programming Language”. In: *Design Thinking Research: Investigating Design Team Performance*. Edited by Christoph Meinel and Larry Leifer. Cham: Springer, 2020, pages 111–131. DOI: [10.1007/978-3-030-28960-7_7](https://doi.org/10.1007/978-3-030-28960-7_7) (cited on page 14).
- [47] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. “Scratch: Programming for All”. In: *Communications of the ACM* 52.11 (Nov. 2009), pages 60–67. ISSN: 0001-0782. DOI: [10.1145/1592761.1592779](https://doi.org/10.1145/1592761.1592779) (cited on page 96).
- [48] Romain Robbes and Michele Lanza. “How Program History Can Improve Code Completion”. In: *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*. ASE ’08. USA: IEEE

- Computer Society, 2008, pages 317–326. DOI: [10.1109/ASE.2008.42](https://doi.org/10.1109/ASE.2008.42) (cited on pages 13, 95).
- [49] Peter Robe and Sandeep Kaur Kuttal. “Designing PairBuddy—A Conversational Agent for Pair Programming”. In: *Transactions on Computer-Human Interaction* 29.4 (May 2022). ISSN: 1073-0516. DOI: [10.1145/3498326](https://doi.org/10.1145/3498326) (cited on page 97).
- [50] Martin P. Robillard. “Automatic Generation of Suggestions for Program Investigation”. In: *SIGSOFT Software Engineering Notes* 30.5 (Sept. 2005), pages 11–20. ISSN: 0163-5948. DOI: [10.1145/1095430.1081711](https://doi.org/10.1145/1095430.1081711) (cited on page 95).
- [51] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. “The Programmer’s Assistant: Conversational Interaction with a Large Language Model for Software Development”. In: *Proceedings of the 28th International Conference on Intelligent User Interfaces. IUI ’23*. Sydney, NSW, Australia: Association for Computing Machinery, 2023, pages 491–514. DOI: [10.1145/3581641.3584037](https://doi.org/10.1145/3581641.3584037) (cited on page 97).
- [52] Mary Beth Rosson and John M. Carroll. “Active Programming Strategies in Reuse”. In: *ECOOP’ 93 — Object-Oriented Programming*. Edited by Oscar M. Nierstrasz. Berlin, Heidelberg: Springer, 1993, pages 4–20. DOI: [10.1007/3-540-47910-4_2](https://doi.org/10.1007/3-540-47910-4_2) (cited on page 12).
- [53] Hesam Samimi, Chris Deaton, Yoshiki Ohshima, Alessandro Warth, and Todd Millstein. “Call by Meaning”. In: *Proceedings of the 2014 International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2014. Portland, Oregon, USA: Association for Computing Machinery, 2014, pages 11–28. DOI: [10.1145/2661136.2661152](https://doi.org/10.1145/2661136.2661152) (cited on page 97).
- [54] David W. Sandberg. “Smalltalk and Exploratory Programming”. In: *SIGPLAN Not.* 23.10 (1988), pages 85–92. ISSN: 0362-1340. DOI: [10.1145/51607.51614](https://doi.org/10.1145/51607.51614) (cited on pages 1, 5, 8).
- [55] Advait Sarkar. “Will Code Remain a Relevant User Interface for End-User Programming with Generative AI Models?” In: *Proceedings of the 2023 SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2023. Cascais, Portugal: Association for Computing Machinery, 2023, pages 153–167. DOI: [10.1145/3622758.3622882](https://doi.org/10.1145/3622758.3622882) (cited on page 100).
- [56] Ben Shneiderman and Catherine Plaisant. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 4th. India: Pearson Education, 2005. ISBN: 978-0-321-19786-3 (cited on page 93).

- [57] Jeremy Singer. “Notes on Notebooks: Is Jupyter the Bringer of Jollity?” In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2020. Virtual, USA: Association for Computing Machinery, 2020, pages 180–186. DOI: [10.1145/3426428.3426924](https://doi.org/10.1145/3426428.3426924) (cited on page 8).
- [58] Joel Spolsky. *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity*. Berkeley, CA, USA: Apress, 2004. ISBN: 978-1-59059-389-9 (cited on page 91).
- [59] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. “IntelliCode Compose: Code Generation Using Transformer”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pages 1433–1443. DOI: [10.1145/3368089.3417058](https://doi.org/10.1145/3368089.3417058) (cited on page 96).
- [60] Marcel Taeumel and Robert Hirschfeld. “Evolving User Interfaces From Within Self-Supporting Programming Environments: Exploring the Project Concept of Squeak/Smalltalk to Bootstrap UIs”. In: *Proceedings of the Programming Experience 2016 Workshop*. PX/16. Rome, Italy: Association for Computing Machinery, 2016, pages 43–59. DOI: [10.1145/2984380.2984386](https://doi.org/10.1145/2984380.2984386) (cited on page 9).
- [61] Marcel Taeumel, Jens Lincke, Patrick Rein, and Robert Hirschfeld. “A Pattern Language of an Exploratory Programming Workspace”. In: *Design Thinking Research: Achieving Real Innovation*. Edited by Christoph Meinel and Larry Leifer. Cham: Springer, Sept. 2022, pages 111–145. DOI: [10.1007/978-3-031-09297-8_7](https://doi.org/10.1007/978-3-031-09297-8_7) (cited on pages 1, 5, 8, 10).
- [62] Marcel Taeumel, Patrick Rein, and Robert Hirschfeld. “Toward Patterns of Exploratory Programming Practice”. In: *Design Thinking Research: Translation, Prototyping, and Measurement*. Edited by Christoph Meinel and Larry Leifer. Springer, Aug. 2021, pages 127–150. DOI: [10.1007/978-3-030-76324-4_7](https://doi.org/10.1007/978-3-030-76324-4_7) (cited on page 8).
- [63] Domenico Talia. “Workflow Systems for Science: Concepts and Tools”. In: *ISRN Software Engineering 2013.1* (Jan. 2013), page 15. DOI: [10.1155/2013/404525](https://doi.org/10.1155/2013/404525) (cited on page 39).
- [64] Steven L. Tanimoto. “A Perspective on the Evolution of Live Programming”. In: *Proceedings of the 1st International Workshop on Live Programming*. LIVE ’13. San Francisco, California: IEEE Press, 2013, pages 31–34. DOI: [10.1109/LIVE.2013.6617346](https://doi.org/10.1109/LIVE.2013.6617346) (cited on page 8).

- [65] Steven L. Tanimoto. “Five Futures with AI Coding Agents”. In: *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming*. Programming ’23. Tokyo, Japan: Association for Computing Machinery, 2023, pages 32–38. DOI: [10.1145/3594671.3594685](https://doi.org/10.1145/3594671.3594685) (cited on pages 92, 100).
- [66] Christoph Thiede. *Symbolic Execution and Applications*. Seminar Report. Hasso Plattner Institute. Mar. 2023. URL: <https://linqlover.github.io/symbolic-execution-survey/report.pdf> (cited on page 12).
- [67] Christoph Thiede and Patrick Rein. *Squeak by Example*. 6.0. Lulu, 2023. ISBN: 978-1-4476-2948-1 (cited on pages 1, 9, 10, 12, 95, 103).
- [68] Christoph Thiede, Willy Scheibel, and Jürgen Döllner. “Bringing Objects to Life: Supporting Program Comprehension through Animated 2.5D Object Maps from Program Traces”. In: *Proceedings of the 19th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*. Volume 1: GRAPP, HUCAPP and IVAPP. IVAPP ’24. INSTICC. Rome, Italy: SciTePress, Feb. 2024, pages 661–669. DOI: [10.5220/0012393900003660](https://doi.org/10.5220/0012393900003660) (cited on page 12).
- [69] Christoph Thiede, Willy Scheibel, Daniel Limberger, and Jürgen Döllner. “Augmenting Library Development by Mining Usage Data from Downstream Dependencies”. In: *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering*. ENASE ’22. INSTICC. SciTePress, 2022, pages 221–232. DOI: [10.5220/0011093700003176](https://doi.org/10.5220/0011093700003176) (cited on pages 10, 95).
- [70] Christoph Thiede, Marcel Taeumel, Lukas Böhme, and Robert Hirschfeld. “Talking to Objects in Natural Language: Toward Semantic Tools for Exploratory Programming”. In: *Proceedings of SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! ’24. To appear. Pasadena, California, USA: Association for Computing Machinery, Oct. 2024. DOI: [10.1145/3689492.3690049](https://doi.org/10.1145/3689492.3690049) (cited on pages ii, 40).
- [71] Christoph Thiede, Marcel Taeumel, and Robert Hirschfeld. “Object-Centric Time-Travel Debugging: Exploring Traces of Objects”. In: *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming*. <Programming>. Tokyo, Japan: Association for Computing Machinery, Mar. 2023, pages 54–60. DOI: [10.1145/3594671.3594678](https://doi.org/10.1145/3594671.3594678) (cited on page 12).
- [72] Christoph Thiede, Marcel Taeumel, and Robert Hirschfeld. “Time-Awareness in Object Exploration Tools: Toward In Situ Omniscient Debugging”. In: *Proceedings of SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*.

Onward! '23. Cascais, Portugal: Association for Computing Machinery, Oct. 2023, pages 89–102. DOI: [10.1145/3622758.3622892](https://doi.org/10.1145/3622758.3622892) (cited on page 12).

- [73] V. Javier Traver. “On Compiler Error Messages: What They Say and What They Mean”. In: *Advances in Human-Computer Interaction 2010* (Jan. 2010). ISSN: 1687-5893. DOI: [10.1155/2010/602570](https://doi.org/10.1155/2010/602570) (cited on page 60).
- [74] David Ungar, Henry Lieberman, and Christopher Fry. “Debugging and the Experience of Immediacy”. In: *Communications of the ACM* 40.4 (Apr. 1997), pages 38–43. ISSN: 0001-0782. DOI: [10.1145/248448.248457](https://doi.org/10.1145/248448.248457) (cited on pages 14, 89, 93).
- [75] Priyan Vaithilingam, Elena L. Glassman, Peter Groenwegen, Sumit Gulwani, Austin Z. Henley, Rohan Malpani, David Pugh, Arjun Radhakrishna, Gustavo Soares, Joey Wang, and Aaron Yim. “Towards More Effective AI-Assisted Programming: A Systematic Design Exploration to Improve Visual Studio IntelliCode’s User Experience”. In: *Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice. ICSE-SEIP '23*. Melbourne, Australia: IEEE Press, 2023, pages 185–195. DOI: [10.1109/ICSE-SEIP58684.2023.00022](https://doi.org/10.1109/ICSE-SEIP58684.2023.00022) (cited on page 96).

Artificial Intelligence and Information Retrieval

- [76] Yuntao Bai et al. *Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback*. 2022. arXiv: [2204.05862](https://arxiv.org/abs/2204.05862) [cs.CL] (cited on page 20).
- [77] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: [2005.14165](https://arxiv.org/abs/2005.14165) [cs.CL] (cited on page 58).
- [78] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. *Sparks of Artificial General Intelligence: Early Experiments with GPT-4*. 2023. arXiv: [2303.12712](https://arxiv.org/abs/2303.12712) [cs.CL] (cited on page 100).
- [79] Hila Chefer, Shir Gur, and Lior Wolf. “Generic Attention-Model Explainability for Interpreting Bi-Modal and Encoder-Decoder Transformers”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision. ICCV '21*. Oct. 2021, pages 387–396. DOI: [10.1109/ICCV48922.2021.00045](https://doi.org/10.1109/ICCV48922.2021.00045) (cited on pages 91, 100).
- [80] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: [2107.03374](https://arxiv.org/abs/2107.03374) [cs.LG] (cited on page 20).

- [81] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Edited by Jill Burstein, Christy Doran, and Tamar Solorio. Volume 1. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pages 4171–4186. DOI: [10.18653/v1/N19-1423](https://doi.org/10.18653/v1/N19-1423) (cited on pages 2, 16, 19).
- [82] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uskoreit, and Neil Houlsby. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. arXiv: [2010.11929](https://arxiv.org/abs/2010.11929) [cs.CV] (cited on pages 16, 18).
- [83] Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. “Precise Zero-Shot Dense Retrieval without Relevance Labels”. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*. Edited by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. Volume 1: Long Papers. Toronto, Canada: Association for Computational Linguistics, July 2023, pages 1762–1777. DOI: [10.18653/v1/2023.acl-long.99](https://doi.org/10.18653/v1/2023.acl-long.99) (cited on page 20).
- [84] Yuan Gong, Yu-An Chung, and James Glass. *AST: Audio Spectrogram Transformer*. 2021. arXiv: [2104.01778](https://arxiv.org/abs/2104.01778) [cs.SD] (cited on page 18).
- [85] Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. “ToolkenGPT: Augmenting Frozen Language Models with Massive Tools via Tool Embeddings”. In: *Advances in Neural Information Processing Systems*. Edited by Alice Oh, Tristian Neumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine. Volume 36. Curran Associates, 2023, pages 45870–45894. arXiv: [2305.11554](https://arxiv.org/abs/2305.11554) [cs.CL] (cited on page 20).
- [86] Barbara Hayes-Roth. “A Blackboard Architecture for Control”. In: *Artificial Intelligence* 26.3 (1985), pages 251–321. ISSN: 0004-3702. DOI: [10.1016/0004-3702\(85\)90063-3](https://doi.org/10.1016/0004-3702(85)90063-3) (cited on page 39).
- [87] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. “The Curious Case of Neural Text Degeneration”. In: *8th International Conference on Learning Representations. ICLR '20*. 2020. arXiv: [1904.09751](https://arxiv.org/abs/1904.09751) [cs.CL] (cited on page 19).
- [88] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu,

and Jürgen Schmidhuber. *MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework*. 2023. arXiv: 2308.00352 [cs.AI] (cited on page 21).

- [89] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search*. 2020. arXiv: 1909.09436 [cs.LG] (cited on pages 19, 95).
- [90] Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. “Large Language Models Are Zero-Shot Reasoners”. In: *Advances in Neural Information Processing Systems*. Edited by Sanmi Koyejo, Shakir Mohamed, Alekh Agarwal, Danielle Belgrave, Kyunghyun Cho, and Alice Oh. Volume 35. Curran Associates, 2022, pages 22199–22213. arXiv: 2205.11916 [cs.CL] (cited on pages 21, 58).
- [91] Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, and Ali Farhadi. *Matryoshka Representation Learning*. 2024. arXiv: 2205.13147 [cs.LG] (cited on page 87).
- [92] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”. In: *Advances in Neural Information Processing Systems*. Edited by Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin. Volume 33. NIPS ’20. Vancouver, Canada: Curran Associates, 2020, pages 9459–9474. ISBN: 978-1-7138-2954-6. arXiv: 2005.11401 [cs.CL] (cited on pages 19, 21, 48, 58).
- [93] Pengfei Li, Jianyi Yang, Mohammad A. Islam, and Shaolei Ren. *Making AI Less “Thirsty”: Uncovering and Addressing the Secret Water Footprint of AI Models*. 2023. arXiv: 2304.03271 [cs.LG] (cited on page 94).
- [94] Lucie Charlotte Magister, Jonathan Mallinson, Jakub Adamek, Eric Malmi, and Aliaksei Severyn. “Teaching Small Language Models to Reason”. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*. Edited by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. Volume 2: Short Papers. Toronto, Canada: Association for Computational Linguistics, July 2023, pages 1773–1781. DOI: 10.18653/v1/2023.acl-short.151 (cited on page 89).
- [95] Yuning Mao, Pengcheng He, Xiaodong Liu, Yelong Shen, Jianfeng Gao, Jiawei Han, and Weizhu Chen. “Generation-Augmented Retrieval for Open-Domain Question Answering”. In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the*

- 11th International Joint Conference on Natural Language Processing. Edited by Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli. Volume 1: Long Papers. Online: Association for Computational Linguistics, Aug. 2021, pages 4089–4100. DOI: [10.18653/v1/2021.acl-long.316](https://doi.org/10.18653/v1/2021.acl-long.316) (cited on page 20).
- [96] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalpanitis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. *Augmented Language Models: A Survey*. 2023. arXiv: [2302.07842](https://arxiv.org/abs/2302.07842) [cs.CL] (cited on page 20).
- [97] Tomáš Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. “Efficient Estimation of Word Representations in Vector Space”. In: *1st International Conference on Learning Representations (ICLR 2013), Workshop Track Proceedings*. Edited by Yoshua Bengio and Yann LeCun. Scottsdale, Arizona, USA, May 2013. arXiv: [1301.3781](https://arxiv.org/abs/1301.3781) [cs.CL] (cited on pages 2, 16, 48).
- [98] Dina Oralbekova, Orken Mamyrbayev, Mohamed Othman, Dinara Kassymova, and Kuralai Mukhsina. “Contemporary Approaches in Evolving Language Models”. In: *Applied Sciences* 13.23 (2023). ISSN: 2076-3417. DOI: [10.3390/app132312901](https://doi.org/10.3390/app132312901) (cited on page 18).
- [99] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: [2303.08774](https://arxiv.org/abs/2303.08774) [cs.CL] (cited on pages 20, 21, 57, 94).
- [100] Long Ouyang et al. “Training Language Models to Follow Instructions with Human Feedback”. In: *Advances in Neural Information Processing Systems*. Edited by Sanmi Koyejo, Shakir Mohamed, Alekh Agarwal, Danielle Belgrave, Kyunghyun Cho, and Alice Oh. Volume 35. Curran Associates, 2022, pages 27730–27744. arXiv: [2203.02155](https://arxiv.org/abs/2203.02155) [cs.CL] (cited on page 20).
- [101] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. *Improving Language Understanding by Generative Pre-Training*. 2018. URL: https://web.archive.org/web/20210126024542/https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf (cited on pages 2, 20).
- [102] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”. In: *The Journal of Machine Learning Research* 21.1 (Jan. 2020). ISSN: 1532-4435. arXiv: [1910.10683](https://arxiv.org/abs/1910.10683) [cs.LG] (cited on page 19).

- [103] Gerard Salton and Christopher Buckley. “Term-Weighting Approaches in Automatic Text Retrieval”. In: *Information Processing & Management* 24.5 (1988), pages 513–523. ISSN: 0306-4573. DOI: [10.1016/0306-4573\(88\)90021-0](https://doi.org/10.1016/0306-4573(88)90021-0) (cited on page 49).
- [104] Hongjin Su, Weijia Shi, Jungo Kasai, Yizhong Wang, Yushi Hu, Mari Ostendorf, Wen-tau Yih, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. *One Embedder, Any Task: Instruction-Finetuned Text Embeddings*. 2023. arXiv: [2212.09741](https://arxiv.org/abs/2212.09741) [cs.CL] (cited on page 82).
- [105] Xiaoyuan Su and Taghi M. Khoshgoftaar. “A Survey of Collaborative Filtering Techniques”. In: *Advances in Artificial Intelligence* 2009.1 (2009). DOI: <https://doi.org/10.1155/2009/421425> (cited on page 50).
- [106] Leonid Nisonovich Vaserstein. “Markovskiye Protsessy Na Schetnom Proizvedenii Prostranstv, Opisyvayushchiye Bol’shiye Sistemy Avtomatov [Markov Processes over Denumerable Products of Spaces, Describing Large Systems of Automata]”. In: *Probl. peredachi inform. [Problems of Information Transmission]* 5.3 (1969). MathSciNet: <http://mathscinet.ams.org/mathscinet-getitem?mr=314115>, pages 64–72 (cited on page 53).
- [107] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. “Attention is All You Need”. In: *Advances in Neural Information Processing Systems*. Edited by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. Vishwanathan, and Roman Garnett. Volume 30. Curran Associates, 2017. arXiv: [1706.03762](https://arxiv.org/abs/1706.03762) [cs.CL] (cited on pages 2, 18, 53).
- [108] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. *Fine-tuned Language Models Are Zero-Shot Learners*. 2022. arXiv: [2109.01652](https://arxiv.org/abs/2109.01652) [cs.CL] (cited on page 21).
- [109] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: [2201.11903](https://arxiv.org/abs/2201.11903) [cs.CL] (cited on page 21).
- [110] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. C. Schmidt. *A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT*. Feb. 2023. arXiv: [2302.11382](https://arxiv.org/abs/2302.11382) [cs.SE] (cited on page 21).
- [111] Hui Yang, Sifu Yue, and Yunzhong He. *Auto-GPT for Online Decision Making: Benchmarks and Additional Opinions*. 2023. arXiv: [2306.02224](https://arxiv.org/abs/2306.02224) [cs.AI] (cited on page 20).

- [112] Wayne Xin Zhao et al. *A Survey of Large Language Models*. 2023. arXiv: 2303.18223 [cs.CL] (cited on pages 2, 21, 91).
- [113] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena". In: *Advances in Neural Information Processing Systems*. Edited by Alice Oh, Tristian Neumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine. Volume 36. Curran Associates, 2023, pages 46595–46623. arXiv: 2306.05685 [cs.CL] (cited on page 58).

