



Talking to Objects in Natural Language

Toward Semantic Tools for Exploratory Programming

Christoph Thiede

christoph.thiede@student.hpi.de
Hasso Plattner Institute
University of Potsdam, Germany

Lukas Böhme

lukas.boehme@hpi.de
Hasso Plattner Institute
University of Potsdam, Germany

Marcel Taeumel

marcel.taeumel@hpi.de
Hasso Plattner Institute
University of Potsdam, Germany

Robert Hirschfeld

robert.hirschfeld@uni-potsdam.de
Hasso Plattner Institute
University of Potsdam, Germany

Abstract

In exploratory programming, programmers often face a semantic gap between their high-level understanding and the low-level interfaces available for interacting with objects in a system. That is, technical object structure and behavior need to be interpreted as abstract domain concepts, which then increases cognitive load and thus impedes exploration progress. We propose *semantic object interfaces* that bridge this gap by enabling contextual, natural-language conversations with objects. Our approach leverages an *exploratory programming agent* powered by a large language model (LLM) to translate natural-language questions into low-level experiments and provide high-level answers. We describe a framework for integrating semantic object interfaces into existing exploratory programming systems, including a prototype implementation in Squeak/Smalltalk using GPT-4o. We showcase the potential of semantic object interfaces through case studies and discuss their feasibility, limitations, and impact on the programming experience. While challenges remain, our approach promises to reduce mental effort and empower programmers to explore and understand systems at a higher level of abstraction for a better programming experience.

CCS Concepts: • Software and its engineering → Integrated and visual development environments; • Human-centered computing → HCI theory, concepts and models; Natural language interfaces.

Keywords: exploratory programming, conversational agents, semantic tools, generative AI, object-oriented programming, natural-language programming, Smalltalk, LLMs, ChatGPT

ACM Reference Format:

Christoph Thiede, Marcel Taeumel, Lukas Böhme, and Robert Hirschfeld. 2024. Talking to Objects in Natural Language: Toward Semantic Tools for Exploratory Programming. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '24)*, October 23–25, 2024, Pasadena, CA, USA. ACM, New York, NY, USA, 17 pages. <https://doi.org/10.1145/3689492.3690049>

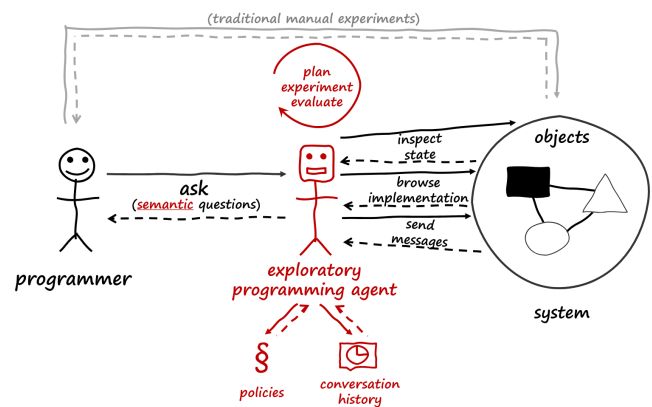


Figure 1. Our approach of *semantic object interfaces* for semantic exploratory programming tools. The programmer expresses high-level, contextual, and often natural-language questions about an object to the interface and receives answers on the same abstraction level. Internally, an exploratory programming agent (red) translates these questions and interacts with the system to perform low-level experiments.

1 Introduction

In the realm of software development, asking questions is a fundamental activity. As programmers want to develop a new feature, fix a bug, or just understand a system, they need to gather a plethora of insights: what is this object about? Where is this data stored? How can I invoke this feature? Why has this cache not been reset? What would it look and feel like if we placed this button there? All this need for



This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Onward! '24, October 23–25, 2024, Pasadena, CA, USA

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1215-9/24/10

<https://doi.org/10.1145/3689492.3690049>

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 [51].

Exploratory programming [41, 46] systems support such conversations to be particularly extensive, vivid, and fast-paced, as programmers can directly interact with systems or their parts through *tools*: for example, a Smalltalk programmer can use an *inspector* to search the internal state of an object, study the source code of a class in a *code browser* to discover helpful interfaces and methods, or evaluate a script in a *workspace* to try out methods and their combinations.

However, every question has its price. 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 version? 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 a significant gap between high-level human intentions and low-level system interactions; in many settings, the conversion is still up to the human side.¹ Thus, exploratory conversations are regularly impeded by mental overheads and distractions as programmers are required to employ logical reasoning and common sense to translate high-level questions into low-level instructions to the system and translate technical results back into their mental model. Consequentially, programmers waste time and lose their flow. In the worst case, the cost of questions is so high that programmers are reluctant to ask them but rely on faulty assumptions instead, renouncing the exploratory paradigm.

While programmers face these challenges day by day, the recent advent of *generative artificial intelligence* (generative AI) has already simplified countless other tasks in various domains. ChatGPT² and other *large language models* (LLMs) [37, 66] have been widely adopted by people and companies for creating, automating, and learning. In particular, programming is experiencing a shift: millions of developers generate common code with the help of GitHub Copilot³,

Tabnine⁴, and others, or they use *conversational agents* such as GitHub Copilot Chat⁵ to explore and modify code bases.

We suggest that programmers can also benefit from generative AI during exploratory programming. Within the exploratory session, programmers could delegate several questions to LLMs that have been trained to process and produce natural-language text and source code and solve problems systematically by generating inner monologue. We assume an object-oriented programming (OOP) context where systems are modeled as objects that interact with each other through messages. This leads to our research question:

How can we support exploratory programming through *semantic object interfaces* that enable *contextual, natural-language conversations* with or about objects?

We propose a notion of *semantic object interfaces* for exploratory programming that allow programmers to answer any emerging questions through natural-language conversations with objects in their programming environment that are powered by *exploratory programming agents*. This leads to the construction of new *semantic tools* that provide conceptual, contextual access to software artifacts, while programmers can remain at a high level of abstraction and do not have to descend into the low-level solution details of code writing or information gathering. We believe that this approach can significantly reduce the cognitive overhead and distractions that programmers face during exploratory programming and empower them to explore and understand systems at a higher level of abstraction.

To examine this idea, we make the following contributions:

1. We describe our approach for semantic object interfaces that allow programmers to talk with objects in their programming system.
2. We design and implement a prototype for this approach in Squeak/Smalltalk through an exploratory programming agent using the LLM GPT-4o.⁶
3. We illustrate the applications of semantic object interfaces through different applications.

The rest of this paper is structured as follows: first, we provide details on the current state of exploratory programming as well as generative AI technologies (section 2). Next, we present our approach for semantic object interfaces that allow programmers to talk with objects in their programming system (section 3). We describe how these can be integrated into exploratory programming systems such as Squeak/Smalltalk (section 4) and implemented through an exploratory programming agent using GPT-4o (section 5). We illustrate the applications of semantic object interfaces through different case studies (section 6), which serve as a

¹High-level and to some extent user-friendly programming languages have come a long way; yet, they remain fundamentally technical and *syntactical* while humans understand each other largely by the mere context and *semantics* of their messages.

²<https://chat.openai.com/>

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

⁴<https://www.tabnine.com/>

⁵<http://archive.today/2024.07.16-010354/https://docs.github.com/en/copilot/using-github-copilot/asking-github-copilot-questions-in-your-ide>

⁶Available on GitHub: <https://github.com/hpi-swa-lab/SemanticSqueak>

base for discussing their general potential and limitations (section 7). Finally, we connect the dots of our approach to existing work (section 8) before summarizing our findings and outlining possible directions for future work (section 9).

2 Background

In this section, we describe our notion of the exploratory programming practice and point out a practical limitation of traditional exploratory programming systems. We provide an overview of current generative AI technologies for programming and sketch how these might be used to advance exploratory programming systems.

2.1 Exploratory Programming

Exploratory programming is an approach to software engineering that promotes a notion of projects where programmers have a rather emergent than upfront understanding of the problem domain [18, 41, 46]. A useful metaphor for this can be found in ordinary science: exploratory programmers apply the scientific method; they iteratively refine their comprehension of both the problem space and the solution space by asking questions and finding answers to them. To find answers, they conduct a substantial number of *experiments*, in which they interact with the system, make observations, deduce, and repeat [51]. In practice, such experiments may manifest through inspecting an object to understand its internal state, sending messages to objects to observe their behavior, or browsing a class to explore its responsibilities⁷.

Exploratory programming systems serve as the foundations for insightful and focused exploratory programming. To facilitate interactive experiments, they offer some degree of *liveness*, which allows programmers to explore and change programs in execution while avoiding longer feedback loops from restarting or compiling the full system [41, 52]. For example, programmers can navigate and configure Linux systems through the Bash shell [39], explore data and algorithms in Jupyter notebooks [49], or develop and debug object-oriented systems in Smalltalk environments [12].

A modern implementation of traditional Smalltalk-80 is *Squeak*, which distinguishes itself by its self-sustained architecture, being highly explorable and malleable [15, 16, 54]: everything is an object (defined by its identity, internal state, and observable behavior), everything happens through message sending between objects, and every object can be inspected and modified dynamically—including classes and methods, call stacks and exceptions, compiler and debugger.

To conduct such live experiments, exploratory programming systems offer several *tools*. In Squeak, one important

tool is the *inspector*, through which programmers can explore any objects in the system. Here, they can view and manipulate an object’s internal state through a sheet of variables and properties; explore object behavior and methods in a *protocol browser*; verify behavior by sending messages through ad-hoc scripts; explore implementation details in context by invoking a *debugger* on any such script execution; browse references to objects and variables through *message traces*; and so on [54, chap. 6]. Thanks to the self-sustained characteristics of Squeak, this toolset is not a rigid black box but can be extended by users to integrate further means for exploration such as program tracing [56, 57], value probing, [38] and runtime visualizations [50, 55].

2.2 A Limitation of Exploratory Programming Systems: Unrestrained Semantic Distances

The effectiveness of exploratory programming rests on the cost of experiments. If programmers feel that the answer to a (simple) question is “too far away” from the question or too difficult to reach, their intuition of questions will be reduced, and they might get frustrated or fall back to guesswork or traditional—but potentially inappropriate, overly complicated—solutions. This observation has been described as the *experience of immediacy* with three dimensions by which tools can reduce these experienced distances [59]:

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 and optimizing efficient algorithms. On the other hand, we argue that semantic distance presents a greater—and largely unmet—challenge to tool developers. This is because the semantic distance does not only occur between inconsistent representations of comparable artifacts but also between an artifact’s representation and its counterpart in the *mental model* of a programmer [32].

The necessary mapping between both technical model and mental model can be described as overall design challenges through the two *gulfs of execution and evaluation* (fig. 2): the gulf of execution references the programmer’s challenges to translate their intentions into inputs to the system’s interface, while the gulf of evaluation represents their challenges to interpret 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, researching interfaces, and writing code, while the gulf of evaluation contains tasks such

⁷In Smalltalk-like systems and this paper, *browsing* refers to the activity of reading the source code of classes and their methods through a structured interface such as a four-pane system browser [12, p. 297ff.]. This is not to be confused with often less focused web browsing.

excerpts from external sources through traditional algorithms (such as database lookups and full-text searches) and include them in the prompt [24].

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 the GPT-4 model, and instructs it to answer the query based on the provided information.⁸ Alternatively, agents can also proactively call functions to retrieve required information. Thus, RAG reduces the challenge of information recall to filtering, contextualizing, and summarizing given text or asking questions about it.

Semantic search: Retrieve information by fetching documents from a corpus based on conceptual instead of lexical similarity using *text embeddings* [40]. This allows filtering and providing only the most relevant information to an LLM, which typically improves the performance of agents (yet might reduce their accuracy) and avoids exceeding the *context window* of text that LLMs can process [64].

Current applications of generative AI for programmers mainly revolve around code generation, refactoring, and searching and navigating code bases or documentation artifacts. Only a few applications use generative technologies to support exploratory programming practices through direct interactions with runtime artifacts or objects. This is a dilemma: programmers can either fall back to traditional, code-centric paradigms—sacrificing the directness and liveness of exploratory programming for the broad support for generative AI tools—, or choose personal exploration over artificial intelligence—having to bridge semantic distances between mental models and tool interactions manually.

We propose *semantic tools* for exploratory programming where programmers can interact with objects in a running system through natural-language conversations, express conceptual questions in terms of their mental model, and get answers from an *exploratory programming agent*, which automatically conducts and encapsulates low-level experiments on its own. With a continually reduced semantic distance, programmers can keep their high-level flow and save precious exploration time for shorter feedback loops and faster insights.

3 Approach: A Semantic Interface for Talking to Objects

We describe our approach toward semantic tools for exploratory programming through a generic *semantic object interface*, which programmers can use to talk with objects in natural language (fig. 1). As part of this interface, high-level and context-dependent *semantic* questions are translated into low-level technical experiments that use traditional object

interfaces (such as state inspection and executable scripts) to provide reasoned answers. Our framework comprises three fundamental actors:

The programmer follows an exploratory practice by asking conceptual questions about an object in a system and expecting answers on a similar level of abstraction. These questions are *semantic*, which means 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 a domain artifact in a (live) system and is defined by its identity, state, and behavior. 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 and may be specified using code, tests, or other forms such as contracts.

Objects can be accessed either by sending them *messages* (which in many languages is also referred to as *invoking methods*) or through *reflection interfaces* that expose their internal state, implementation, or location in the object graph (e.g., for retrieving all instance variables of a class or all pointers to an object from other objects).

The exploratory programming agent is a 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 their results into answers for the programmer.

Internally, the agent processes the programmer’s question, develops a research 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 question.

The agent uses two resources: a set of *policies* and a *conversation history*. *Policies* define abstract agent behavior, such as the kind 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.

Our framework allows programmers to ask arbitrary questions about objects that 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?”, or “What is the age distribution of weekend customers?”.

⁸Microsoft. 2023-11-21. *How Copilot Works*. <https://web.archive.org/web/20240530235213/https://www.microsoft.com/en-us/bing/do-more-with-ai/how-bing-chat-works?form=MA13KP>

Epistemic questions (or “how” questions) refer to the behavior of objects, domain concepts, and their implementation. Programmers ask these questions 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 modeled?”, or “How can we analyze the shopping behavior of customers?”.

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

State inspection allows the internal information of objects to be explored, for instance, by looking up a variable or enumerating all properties.

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 for understanding the usage of objects by example. *Documentation* can be provided through comments, examples, or alternative system descriptions.

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.⁹

Note that the agent generally does not require manual preparation for specific systems and packages from domain experts or programmers but will learn about them on its own. Thus, even to answer simple functional questions, it 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-send to an unknown system is preceded by auto-browsing its implementation, through which the agent discovers the relevant protocols and messages to use.

For example, when a programmer asks “Which product has generated the highest profit?” about a selected shop object, the agent will first execute 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»items`, and `Product»price`, the agent will then plan how to combine this information to compute the most profitable product and run a script that queries the system for this information as another experiment. Eventually, the agent

⁹From 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.

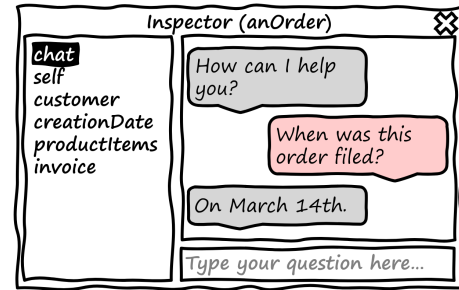


Figure 3. Possible integration of a conversational semantic interface into a traditional object inspection tool. Programmers can ask conceptual questions about objects in natural languages besides inspecting their internal state.

will evaluate the results of this experiment and return a summarized answer in natural language. The question was answered and the programmer can continue by asking another question or performing another exploratory activity.

Thus, semantic object interfaces support programmers in asking conceptual questions about objects—directly and without unnecessary mappings—, which an exploratory programming agent then automatically interprets, translates into technical experiments, and executes those to answer the questions. In the next two sections, we describe our ideas for integrating semantic object interfaces into exploratory programming systems and describe our implementation of an exploratory programming agent using GPT-4o.

4 Integrating Semantic Interfaces into an Exploratory Programming System

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*.

4.1 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. Our new *conversation mode* in inspection tools allows programmers to ask semantic questions about an object. Thus, in addition to technical state inspection, they can then also (naturally) chat about the object with the agent (fig. 3).

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

4.2 Semantic Messaging in Scripts

Scripting can be a 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 a language extension for object-oriented 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, e.g.:

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

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 or implement algorithms for complex problems.

4.3 Implementation in Squeak/Smalltalk

We implemented a prototype⁶ of our proposed semantic object interfaces for the Squeak/Smalltalk programming system using our conversational agent framework SEMANTICTEXT¹⁰. 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, which inserts a new chat item into the inspector’s field list that references a conversation with our exploratory programming agent, and which embeds a minimal version of the SEMANTICTEXT chat interface for the conversation when this field is selected (fig. 4).

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: [15, sec. 5.11] and forwarding all unknown messages to the agent. Through an additional policy (see

¹⁰<https://github.com/hpi-swa-lab/Squeak-SemanticText>

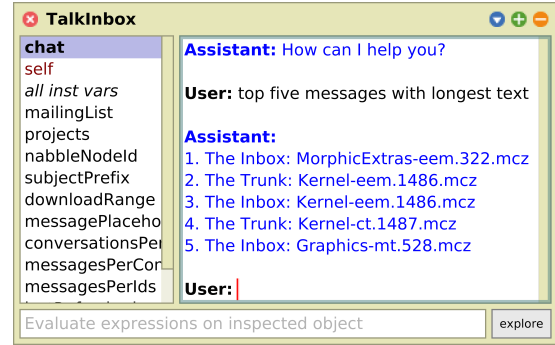


Figure 4. Our integration of a semantic conversational interface into Squeak’s inspector (here: “chat” item in the field list on the left). In this example, the user chats with an archive of the Squeak mailing list¹¹ to find exceptionally large posts.

section 3), 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 of processing semantic messages:

Semantic proxies, which are constructed explicitly and override #doesNotUnderstand: to handle messages:

```
self semanticProxy mostOftenBoughtArticle.
aProduct semanticProxy countSalesTo: bob.
```

The ? and ! operators, which 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 reduced typing effort.

5 Building an Exploratory Programming Agent with GPT-4o

We implemented our exploratory programming agent through a conversational autonomous agent using OpenAI’s LLM GPT-4o¹², 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 [33], and our conversational agent framework SEMANTICTEXT for Squeak. We implement the agent’s policies through prompt engineering and map the system interface to a set of callable functions.¹³

¹¹<https://github.com/hpi-swa-lab/squeak-inbox-talk>

¹²Used in the version gpt-4o-2024-05-13.

¹³We provide the full conversation including all system prompts and function calls for the example from fig. 6 in our GitHub repository (<https://github.com/hpi-swa-lab/SemanticSqueak/blob/11637e5/assets/Text.conversation>).

5.1 Implementing Policies through Prompts

We define a set of policies that 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 user in exploring an object in a system. In conversation mode (section 2.3), the *assistant* shall also identify as the explored object itself, allowing users to talk to the object in the second person for increased semantic immediacy.¹⁴

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. When referring to object protocols, the agent shall automatically browse the implementation or senders of messages whose behavior is unclear. Additionally, the agent shall automatically test message-sends by executing them before suggesting them to the user.

Output format: The agent shall provide brief and concise answers and avoid full sentences unless requested otherwise. When addressed through semantic messages, it shall answer an object instead of natural-language text. Objects can be fetched from the system or be created by the agent; primitive values (such as numbers or 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. 5). While we iterated our prompt design several times, we did not conduct a systematic, evaluation-based approach to prompt engineering [67]; 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 (“bootstrapping the exploration” in fig. 5).

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, we provide a brief situational explanation and point the agent to relevant protocols for querying the entire debugger stack.

¹⁴This 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).

This pre-generated conversation prefix serves several purposes: first, it provides a 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 [20], few-shot prompting [6], and retrieval-augmented generation [24].

5.2 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 programming systems. For example, the agent can *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 packages, classes, and methods; or *send messages*

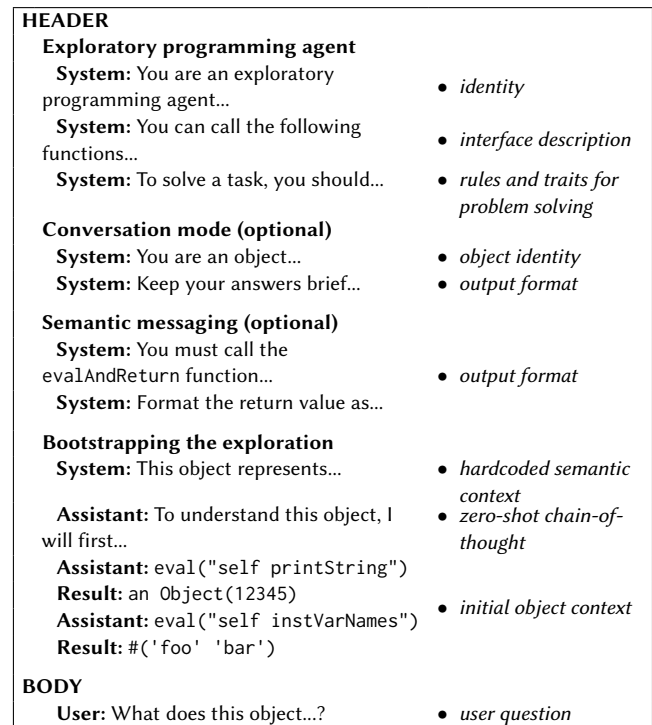


Figure 5. Schematic prompt design for conversations of the exploratory programming agent with user and system. We translate policies for the identity, strategies, and output formats of the agent to detailed instructions. Through pre-generated assistant messages and function calls, we stimulate it to engage in inner monologue and several experiments.

Table 1. The function interface that connects our exploratory programming agent to the 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 (for static methods), 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 subset. (The subset is probabilistically sampled from an ad-hoc clustering of all senders to optimize for the uniqueness and diversity of results. If an optional query argument was provided, sampling is weighted for the relevance of methods using the fifth power of the distance between cluster centers and the query. Clusters and relevance are determined based on source code embeddings using OpenAI’s text-embedding-3-large model with 256 dimensions.)

to objects by evaluating scripts. [Table 1](#) shows the function interface of our prototypical agent for a Smalltalk system. Below, we discuss two challenges in designing these functions: the granularity of function calls and the programming language proficiency of the agent.

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 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 (as 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 [58]: 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")`¹⁵, we offer a dedicated `browseClass()` function for this purpose instead ([table 1](#)).

To avoid dangerous side effects of AI-generated code such as data loss or system crashes, it is also possible to evaluate all requested scripts in a lightweight sandbox or only apply their side effects to the system after manual review.¹⁶ However, we only observed a small number of incidents during our experiments and have not installed such a mechanism in our prototype to avoid additional complexity in the agent due to managing “different realities” [61].

6 Semantic Object Interfaces by Example

In the following, we illustrate possible applications of semantic object interfaces in exploratory programming systems through two different case studies: understanding text formatting in Squeak and extending programming tools.

6.1 Exploring Text Formatting in Squeak

In the first example, we describe how a programmer uses a conversational interface to explore Squeak’s API 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, our programmer discovers an existing Text object in the system that looks like this:

`ABCDE`¹⁷

To understand its design and behavior related to formatting, the programmer invokes an inspector on this text ([fig. 6](#)).

¹⁵A similar technique is used in the current version of ChatGPT Plus when browsing long documents.

¹⁶<https://github.com/LinqLover/SimulationStudio>

¹⁷Alternative description for accessibility: The first three letters are emphasized in bold, and the last letter is underlined.

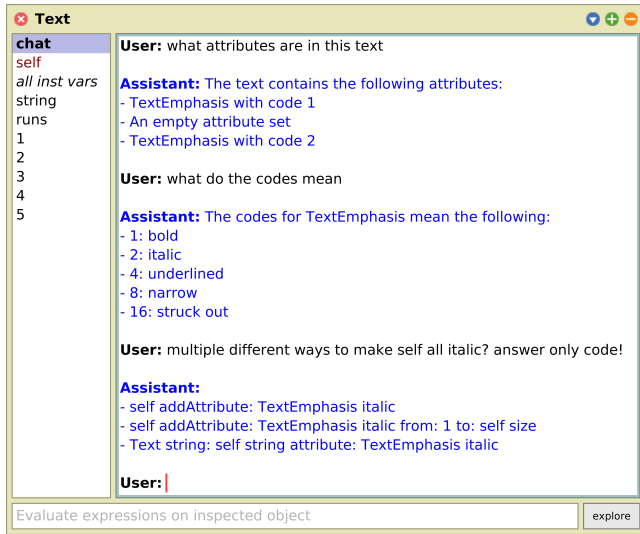


Figure 6. Using the semantic conversation mode in an inspector to chat with a formatted `Text` object, understand the attributes it uses for formatting, and explore the available protocols for applying other formats to texts.

First, they wonder what attributes are contained in the text. As the internal structure of nested collections looks slightly overwhelming, they switch to the chat interface 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 sending 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 different code examples to italicize the entire object. In response, the agent automatically browses the protocols of the `Text` class, 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 on their own, or integrate them into their own program.

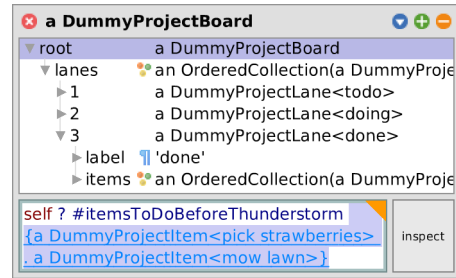


Figure 7. Sending a semantic message to search and filter a small project management system for contextual questions. The agent automatically explores the available protocols of the project management systems to retrieve items from the lanes of the project board and then filters them.

Thus, programmers can use the conversation mode in inspectors to ask both functional and epistemic questions about objects, which the agent attempts to answer by extracting, analyzing, and synthesizing information. Through functional questions, programmers can access, search, or summarize domain information. Another example of functional questions is filtering items in a task management system based on their content (fig. 7). Through epistemic questions, they 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.

6.2 Toward a Semantic Toolset for Exploratory Programming

Here, we describe another application of semantic object interfaces 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 [9]. 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, or Smalltalk systems [35].

We propose a simple mechanism for OOUI frameworks that integrates semantic conversation interfaces with the visual mapping of OOUIs. This allows users to click on domain objects on their screen to talk to them in natural language, while not requiring domain developers to put manual effort into writing prompts or preparing contextual information for LLMs (the agent still fetches all required information by itself through automated experiments). For example, in a project management system that organizes task items in

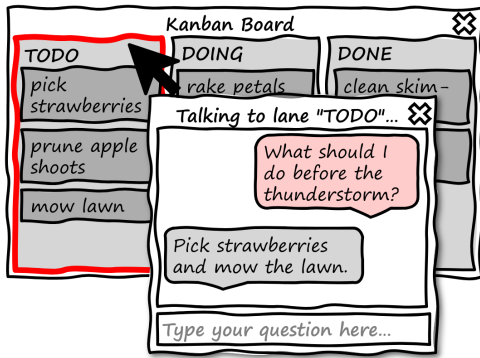


Figure 8. Through the generic integration of semantic object interfaces into object-oriented user interfaces, users can select arbitrary domain objects from the UI and start a natural-language conversation with them. Here, a user talks to a lane in a project management system to filter its items.

boards and lanes, a user could select a lane and ask for a semantic selection or summary of the items it contains (fig. 8).

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 their derived artifacts 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. 9), explain or refactor a code block (fig. 10), search for the origin of values or cause of state changes in a program stack or slice (fig. 11), identify the bottlenecks in a program trace, etc.

Thus, we effectively upgrade existing programming tools to *semantic tools* by extending them with a conversational interface for an agent, which 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.

7 Discussion

In the following, we discuss the feasibility and limitations of semantic object interfaces as well as their possible impact on the experience of exploratory programmers.

Over the past six months, we have successfully used our prototype for several tasks of a similar size to that of the examples in fig. 4 and section 6. At the same time, semantic object 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 for the exploratory programming agent 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 LLMs and systematic prompt engineering or fine-tuning.

7.1 Feasibility and Limitations

We report on the accuracy and performance of exploratory programming agents from experiments with our prototype.

Accuracy of exploratory programming agents. 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 (e.g., to “make up facts”): for instance, our agent was unable in some cases to write executable scripts based on given source code or error messages, and it would sometimes 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

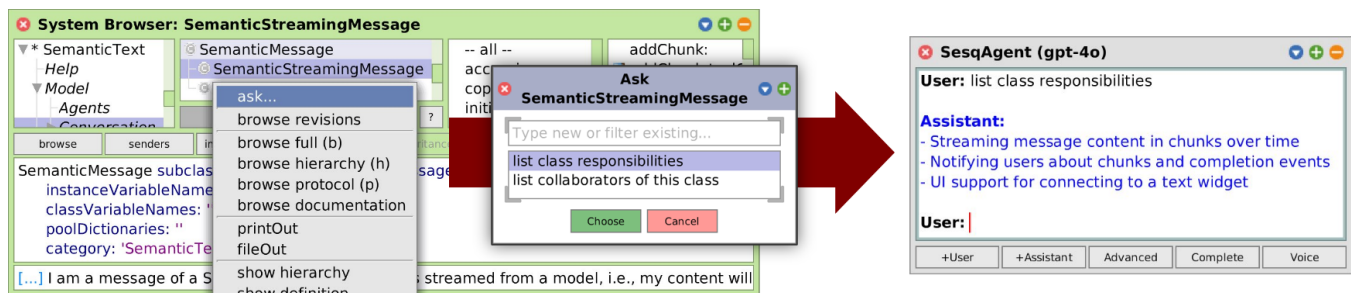


Figure 9. In a semantic code browser (here: the default system browser of Squeak 6.1Alpha), programmers can engage in natural-language conversations with classes to explore them, e.g., by asking for their responsibilities or collaborators.

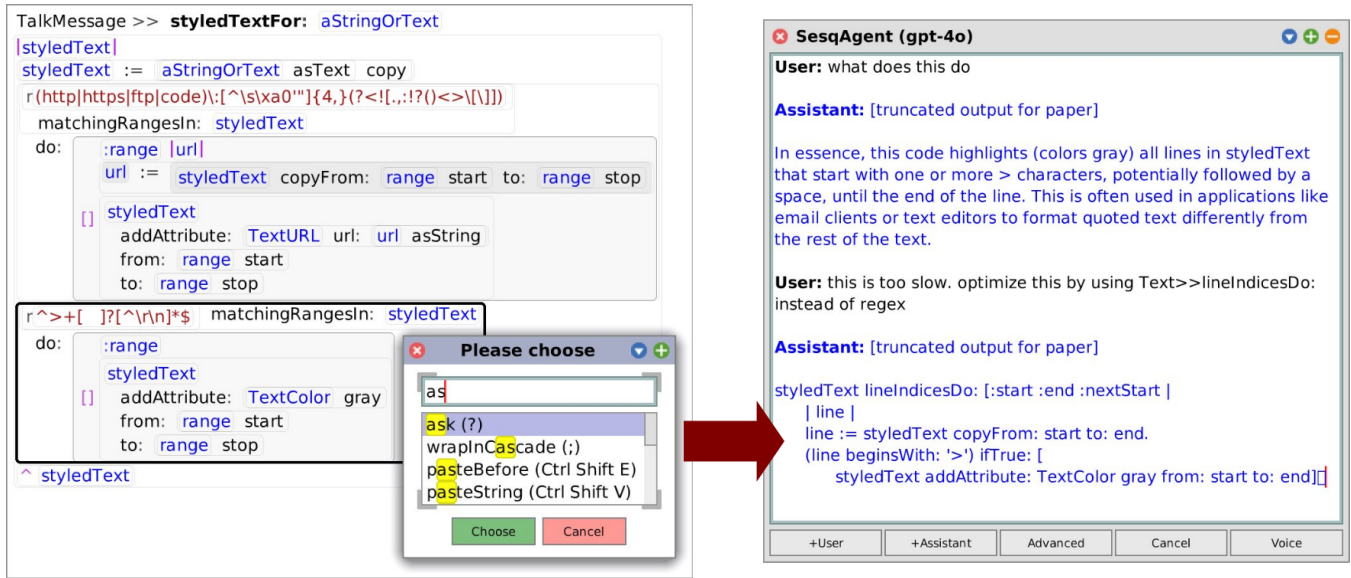


Figure 10. In a semantic projectional editor (here: Sandblocks [5]), programmers can chat with single code blocks to explain, refactor, or execute them.

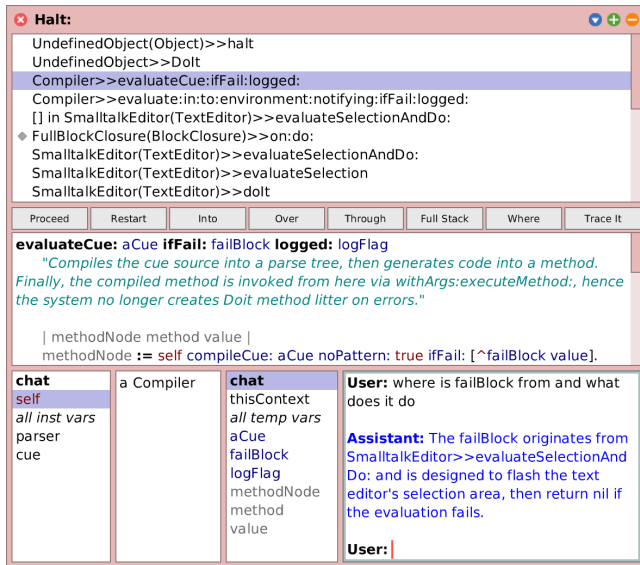


Figure 11. In a semantic debugger (here: the Squeak debugger), programmers can ask for the origin and meaning of values on the program stack.

version control system because it would be “not allowed to provide confidential information”.

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 paper. However, we can illustrate its current state through a few anecdotal examples: for the questions about the Text from section 6.1, 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 (fig. 11), 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 write 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 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). 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, or it would mix 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 abilities could be further improved.¹⁸ Alternatively, even a more systematic approach to prompt engineering with extended chain-of-thought or few-shot prompts might improve these capabilities.

Performance. In our setting with a cloud-based LLM service, resource consumption mainly manifests as latencies in the agent’s experiments and responses and as monetary costs for using the API.

Response times 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. 6), 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 some cases. By streaming responses character-wise from the API, we reduce the delay to the start of the answer by factors between 20 % and 80 %.

Similarly, monetary costs vary depending on the complexity of questions and answers, ranging from \$0.10 (US dollars) 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 [21]; following Jevon’s paradox, the actual cost could be many times higher as programmers might become negligent of their expenses.¹⁹

¹⁸For comparison, ChatGPT’s code interpreter shows a higher proficiency in writing Python code, using Python 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.

¹⁹We also experimented with OpenAI’s gpt-3.5-turbo model instead of gpt-4o-2024-05-13. While this would improve response times and even 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.

Considering the ongoing trend of more powerful, faster, and less energy-consuming LLMs being developed, we assume that the latencies and expenses for invoking our exploratory programming agent could become small enough to no longer pose a practical barrier for programmers and organizations with more affluent budgets. For illustration, while we were working on this approach and paper over six months, OpenAI’s respective flagship model (from gpt-4-0613 to gpt-4o-2024-05-13) 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 are viewing *small language models* as a promising alternative to cloud-based LLMs, since they could be trained to handle particular tasks such as exploratory programming more competently and efficiently and run on commodity hardware with reduced response times [27].

7.2 Programming Experience

From experiments with our prototype, we report and discuss preliminary insights into the programming experience in semantic exploratory programming tools.

Abstraction level. Semantic object interfaces allow programmers to express their questions on a high abstraction level while delegating low-level experiments to the exploratory programming system. Thus, distractions and cognitive load are reduced and they can better maintain flow [10]. This abstraction can also 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 traditional exploratory programming manner, programmers might encounter other protocols by chance that give them a peripheral impression of a system’s domain concepts and capabilities, or even find an unanticipated method that directly solves their overarching problem. As semantic tools for exploration reduce such manual experiments, programmers will make fewer such serendipitous discoveries.

Natural-language interface. Not being required to express their intentions and questions in a formal (programming) language, programmers are encouraged to ask questions “as they come through their mind”, lowering the gulf of execution [32] and contributing to the experience of immediacy [59]. To ask follow-up questions, they do not have to repeat or modify existing inputs but can express them in the semantic context of an existing conversation. Finally, answers can be customized and individualized based on the intentions and preferences of programmers; for example, novice programmers could ask the agent to always explain

their proposed solutions step-by-step, while experts might prefer concise outputs.

Delegation of control. Remaining on a conceptual level of abstraction also involves delegating control to another instance, being the exploratory programming agent in our approach. If responsibilities are clearly separated and experienced temporal immediacy is maintained (see next paragraph), this can be an effective division of labor. However, as contemporary LLMs—and thus the agents based on them—are still prone to err or fail, programmers’ trust in the agent might be reduced after some time. If the agent errs or fails or programmers suspect it thereof, they have to intervene and fall back to traditional low-level practices and switch abstraction levels. During this, they have to either understand the previous attempts of the agent and correct them or discard the agent’s work and make most of the effort again.²⁰

Impact of performance limitations. A noticeable resource consumption (section 7.1) 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 [48, p. 473]; the delay of some complex answers, though, may challenge the patience of programmers, until LLMs are further accelerated. Monetary costs, on the other hand, led to a psychological fear of expenses in our experiments (“How expensive will this question be? Is the answer really worth two dollars?”). We tried to mitigate this effect by displaying a prominent expense watcher in the UI and defining quotas for each semantic question but couldn’t completely shake off the uneasy feeling whenever we wished to talk to an object.

Note that in our current prototypes, we have not made any attempts to reduce resource consumption by systematically minimizing prompts or tool interfaces. As hinted above, we see several opportunities for the future.

7.3 Ethical Considerations

We see several ethical concerns regarding the use of (cloud-based) LLMs such as concentration of economic and political power, high energy intake and water consumption (a typical question in our prototype might use 0.05 kWh and between 50 ml and 750 ml water [25]²¹), potentially unsafe biases impacting the accessibility, internationalization, or security of data analyses or recommended solutions [33], and the training of models based on the intellectual property of unconsulted content creators and labor of inadequately

²⁰Our current prototype already provides an option for displaying all experiments and thought processes of the agent as part of the conversation, allowing programmers to verify its answers. Still, we note the intrinsic complexity of understanding the agent’s steps, which obstructs our vision of a clear division of labor between human and machine.

²¹Actual numbers might vary because the only data in [25] leaves room for interpretation, depends on the region of the compute clusters, and resource consumption might have changed with the development of newer models.

provided click workers²². Before adopting semantic object interfaces in practice, it would be advisable to consider these implications and evaluate possible alternatives such as small language models or open-source LLMs.

8 Related Work

Here, we discuss related work in the areas of conversational interfaces for exploratory programming and other AI-based programming tools. Our concept of an exploratory programming agent to which the programmer can delegate tasks shares similarities to a pair-programming setup [4] 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, semantic object interfaces are the first work to support the general exploratory programming practice by providing conceptual access to runtime objects.

Question-based debugging tools. Different tools have been proposed to support programmers during debugging sessions by answering high-level questions [11, 31]. For example, the WHYLINE allows programmers to ask questions about the causes of certain events in their program [19]. 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 with the system.

CHATDBG provides an LLM-based agent to answer natural-language questions about an errored program in a debugger [23]. 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 [1] or OPENUI²³ facilitate iterative prototyping of visualizations and user interfaces through natural-language instructions. SPELLBURST also offers a node-based visual programming interface for managing derivations and alternatives of prototypes.

Code-centric AI tools. Recent advancements in generative AI have led to the development of various code-centric AI tools that aim to support programmers in their tasks. First, code completion tools such as GitHub Copilot³ and Tabnine⁴ provide suggestions for code snippets based on the context of the current code. Programmers can use such tools to accelerate their coding process and explore different interfaces and solution approaches [3]. Second, conversational agents such

²²Billy Perrigo. 2023-01-18. *OpenAI Used Kenyan Workers on Less Than \$2 per Hour to Make ChatGPT Less Toxic*. Time. <https://web.archive.org/web/20240704034409/https://time.com/6247678/openai-chatgpt-kenya-workers/>

²³<https://github.com/wandb/openui>

as ChatGPT and GitHub Copilot Chat⁵ allow programmers to ask questions about their code bases and other interfaces, to understand and fix bugs based on their code, or to request code changes in natural language [22, 43, 44].

Natural-language programming. 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 [30]. Different metaphors such as tools²⁴ and operating systems [28] were discussed to structure natural-language directives to scale 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 [45]. While component developers and users are still required to express intentions in a formal language to maintain and access a detailed specification, this approach enables programmers to communicate with software systems in a semantic, interface-agnostic style.

9 Conclusion and Future Work

In this paper, we have proposed semantic object interfaces, which allow programmers to interact with objects in their programming environment through high-level, natural-language, contextual questions. This fosters our vision of semantic exploratory programming systems in which programmers can remain in their conceptual mental model while exploring and prototyping systems. We have described the design of a framework for semantic object interfaces using an exploratory programming agent that takes semantic questions, plans experiments, and executes them by interacting with the system through programmatic interactions. We have sketched how semantic object interfaces can be integrated into existing programming systems through a conversation mode in object inspection tools and a language extension for sending semantic messages to objects. Although we have presented a prototypical implementation of our approach in Squeak/Smalltalk using GPT-4o, it can be applied to any interactive programming system that supports dynamic code execution and basic reflective access to runtime objects such as computational notebooks [49], the Lively Kernel [17, 26], and a debugger for Python.

Our discussion of the approach has shown that semantic object interfaces can be used to answer a wide range of functional and epistemic questions about objects and improve the semantic immediacy of different programming tools, and we

noted conceptual and implementational challenges, including the limited accuracy and performance of current LLMs and reduced serendipitous discoveries by programmers.

For future work, we see two important directions: agent capabilities and human-agent interactions. First, the capabilities of exploratory programming agents for problem solving, especially by following exploratory programming practices, and their performance need to be improved. This involves general advancements in training, scaling, and optimizing LLMs as well as tuning prompts or fine-tuning models for exploratory practices.

Second, we aim to improve the interactions between programmers and semantic object interfaces to allow programmers to cooperate with agents rather than delegating tasks to them. For example, we want to facilitate the comprehension and reuse of agents' experiments, thus making it easier for programmers to intervene in the agents' work and creating flexibility in combining manual and delegated exploratory activities. By providing further semantic context to the agent such as previous conversations and manual experiments of programmers, the naturalness of semantic object interfaces could be improved. This also includes possible models of separate responsibilities where agents could ask questions back to programmers to help clarify goals and system specifications. Finally, we envision new interfaces such as shared artifact spaces between agents and programmers to seamlessly integrate such AI agents into the exploratory process.

Much about the future of programming in an era of evolving AI technologies and possibly artificial general intelligence [7] remains uncertain [53]. While it is likely that low-level programming tasks will be further automated, analyzing domains and conceptualizing possible solutions have proven as 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 [47] and beyond, semantic programming 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.

Acknowledgments

We are very grateful to Toni Mattis for educating us about the technical foundations of LLMs, u/plasticpears for discussing their vision of AI-augmented exploratory programming systems with us²⁵, and the anonymous reviewers for contributing their insightful and detailed comments.

References

- [1] Tyler Angert, Miroslav Suzara, Jenny Han, Christopher Pondoc, and Hariharan Subramonyam. 2023. Spellburst: A Node-based Interface for

²⁴GPTSCRIPT: <https://github.com/gptscript-ai/gptscript>

²⁵https://www.reddit.com/r/smalltalk/comments/1b3dx4q/smalltalk_llms

- Exploratory Creative Coding with Natural Language Prompts. In *Proceedings of the 36th Annual Symposium on User Interface Software and Technology* (San Francisco, CA, USA) (UIST '23). ACM, New York, NY, USA, Article 100, 22 pages. <https://doi.org/10.1145/3586183.3606719>
- [2] Yuntao Bai et al. 2022. Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback. arXiv:2204.05862 [cs.CL]
- [3] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages* 7, OOP-SLA1, Article 78 (4 2023), 27 pages. <https://doi.org/10.1145/3586030>
- [4] Kent Beck. 2000. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, USA.
- [5] Tom Beckmann, Stefan Ramson, Patrick Rein, and Robert Hirschfeld. 2020. Visual Design for a Tree-Oriented Projectional Editor. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming* (Porto, Portugal) (<Programming> '20 Companion). ACM, New York, NY, USA, 113–119. <https://doi.org/10.1145/3397537.3397560>
- [6] Tom B. Brown et al. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]
- [7] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrmke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. arXiv:2303.12712 [cs.CL]
- [8] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]
- [9] Dave Collins and Dave Collins. 1995. *Designing Object-Oriented User Interfaces*. Benjamin Cummings, Redwood City, CA.
- [10] Mihaly Csikszentmihalyi. 2008. *Flow: The Psychology of Optimal Experience* (1 ed.). Harper Perennial, New York.
- [11] Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers. 2011. 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* (Kongens Lyngby, Denmark) (PPPJ '11). ACM, New York, NY, USA, 71–80. <https://doi.org/10.1145/2093157.2093168>
- [12] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, USA.
- [13] Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. 2023. ToolkenGPT: Augmenting Frozen Language Models with Massive Tools via Tool Embeddings. In *Advances in Neural Information Processing Systems*, Alice Oh et al. (Eds.), Vol. 36. Curran Associates, 45870–45894. arXiv:2305.11554 [cs.CL]
- [14] Sirui Hong et al. 2023. MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework. arXiv:2308.00352 [cs.AI]
- [15] Daniel Ingalls. 2020. The Evolution of Smalltalk: From Smalltalk-72 through Squeak. *Proceedings of the ACM on Programming Languages* 4, HOPL, Article 85 (6 2020), 101 pages. <https://doi.org/10.1145/3386335>
- [16] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Atlanta, Georgia, USA). *SIGPLAN Not.* 32, 10, 318–326. <https://doi.org/10.1145/263700.263754>
- [17] Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. 2008. The Lively Kernel: A Self-Supporting System on a Web Page. In *Self-Sustaining Systems*, Robert Hirschfeld and Kim Rose (Eds.). Springer, Berlin, Heidelberg, 31–50.
- [18] Mary Beth Kery and Brad A. Myers. 2017. Exploring Exploratory Programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Raleigh, NC, USA, 25–29. <https://doi.org/10.1109/VLHCC.2017.8103446>
- [19] Amy J. Ko and Brad A. Myers. 2004. Designing The Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria) (CHI '04). ACM, New York, NY, USA, 151–158. <https://doi.org/10.1145/985692.985712>
- [20] Takeshi Kojima et al. 2022. Large Language Models are Zero-Shot Reasoners. In *Advances in Neural Information Processing Systems*, Sanmi Koyejo et al. (Eds.), Vol. 35. Curran Associates, 22199–22213. arXiv:2205.11916 [cs.CL]
- [21] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. 2018. The Road to Live Programming: Insights from the Practice. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) (ICSE '18). ACM, New York, NY, USA, 1090–1101. <https://doi.org/10.1145/3180155.3180200>
- [22] Kimio Kuramitsu, Yui Obara, Miyu Sato, and Momoka Obara. 2023. 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) (SPLASH-E 2023). ACM, NY, USA, 50–59. <https://doi.org/10.1145/3622780.3623648>
- [23] Kyla Levin, Nicolas van Kempen, Emery D. Berger, and Stephen N. Freund. 2024. ChatDBG: An AI-Powered Debugging Assistant. arXiv:2403.16354 [cs.SE] <https://arxiv.org/abs/2403.16354>
- [24] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Kuttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems* (Vancouver, Canada) (NIPS '20, Vol. 33), Hugo Larochelle et al. (Eds.). Curran Associates, Red Hook, NY, USA, Article 793, 16 pages.
- [25] Pengfei Li, Jianyi Yang, Mohammad A. Islam, and Shaolei Ren. 2023. Making AI Less “Thirsty”: Uncovering and Addressing the Secret Water Footprint of AI Models. arXiv:2304.03271 [cs.LG]
- [26] Jens Lincke, Patrick Rein, Stefan Ramson, Robert Hirschfeld, Marcel Taeumel, and Tim Felgentreff. 2017. Designing a Live Development Experience for Web-Components. In *Proceedings of the 3rd SIGPLAN International Workshop on Programming Experience* (Vancouver, BC, Canada) (PX/17.2). ACM, New York, NY, USA, 28–35. <https://doi.org/10.1145/3167109>
- [27] Lucie Charlotte Magister et al. 2023. Teaching Small Language Models to Reason. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.), Vol. 2: Short Papers. Association for Computational Linguistics, Toronto, Canada, 1773–1781. <https://doi.org/10.18653/v1/2023.acl-short.151>
- [28] Kai Mei et al. 2024. AIOS: LLM Agent Operating System. arXiv:2403.16971 [cs.OS]
- [29] Grégoire Mialon et al. 2023. Augmented Language Models: A Survey. arXiv:2302.07842 [cs.CL]
- [30] Rada Mihalcea, Hugo Liu, and Henry Lieberman. 2006. NLP (Natural Language Processing) for NLP (Natural Language Programming). In *Computational Linguistics and Intelligent Text Processing*, Alexander Gelbukh (Ed.). Springer, Berlin, Heidelberg, 319–330.
- [31] Brad A. Myers, John F. Pane, and Amy J. Ko. 2004. Natural Programming Languages and Environments. *Commun. ACM* 47, 9 (9 2004), 47–52. <https://doi.org/10.1145/1015864.1015888>
- [32] Donald A. Norman. 1986. *Cognitive Engineering*. Lawrence Erlbaum Associates, Hillsdale, NJ, USA, 31–61.
- [33] OpenAI et al. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
- [34] Long Ouyang et al. 2022. Training Language Models to Follow Instructions with Human Feedback. In *Advances in Neural Information Processing Systems*, Sanmi Koyejo et al. (Eds.), Vol. 35. Curran Associates, 27730–27744. arXiv:2203.02155 [cs.CL]
- [35] Richard Pawson and Robert Matthews. 2001. Naked Objects: A Technique for Designing More Expressive Systems. *SIGPLAN Not.* 36, 12

- (12 2001), 61–67. <https://doi.org/10.1145/583960.583967>
- [36] Chen Qian et al. 2023. Communicative Agents for Software Development. arXiv:2307.07924 [cs.SE]
- [37] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving Language Understanding by Generative Pre-Training. <https://openai.com/research/language-unsupervised>
- [38] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-Style Programming: Design and Implementation of an Integration of Live Examples into General-Purpose Source Code. *The Art, Science, and Engineering of Programming* 3, 3, Article 9 (2 2019), 39 pages. <https://doi.org/10.22152/programming-journal.org/2019/3/9>
- [39] Eric Steven Raymond. 2003. *The Art of UNIX Programming*. Addison-Wesley, Boston.
- [40] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. arXiv:1908.10084 [cs.CL]
- [41] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2019. Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness. *The Art, Science, and Engineering of Programming* 3, 1 (07 2019), 33 pages. <https://doi.org/10.22152/programming-journal.org/2019/3/1>
- [42] Patrick Rein, Marcel Taeumel, and Robert Hirschfeld. 2020. Towards Empirical Evidence on the Comprehensibility of Natural Language Versus Programming Language. In *Design Thinking Research: Investigating Design Team Performance*, Christoph Meinel and Larry Leifer (Eds.). Springer, Cham, 111–131. https://doi.org/10.1007/978-3-030-28960-7_7
- [43] Peter Robe and Sandeep Kaur Kuttal. 2022. Designing PairBuddy—A Conversational Agent for Pair Programming. *Transactions on Computer-Human Interaction* 29, 4, Article 34 (5 2022), 44 pages. <https://doi.org/10.1145/3498326>
- [44] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. 2023. 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* (Sydney, NSW, Australia) (IUI '23). ACM, New York, NY, USA, 491–514. <https://doi.org/10.1145/3581641.3584037>
- [45] Hesam Samimi, Chris Deaton, Yoshiki Ohshima, Alessandro Warth, and Todd Millstein. 2014. Call by Meaning. In *Proceedings of the 2014 International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, Oregon, USA) (Onward! 2014). ACM, New York, NY, USA, 11–28. <https://doi.org/10.1145/2661136.2661152>
- [46] David W. Sandberg. 1988. Smalltalk and Exploratory Programming. *SIGPLAN Not.* 23, 10 (1988), 85–92. <https://doi.org/10.1145/51607.51614>
- [47] Advait Sarkar. 2023. 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* (Cascais, Portugal) (Onward! 2023). ACM, New York, NY, USA, 153–167. <https://doi.org/10.1145/3622758.3622882>
- [48] Ben Shneiderman and Catherine Plaisant. 2005. *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (4th ed.). Pearson Education, India.
- [49] Jeremy Singer. 2020. Notes on Notebooks: Is Jupyter the Bringer of Jollity?. In *Proceedings of the 2020 SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Virtual, USA) (Onward! 2020). ACM, New York, NY, USA, 180–186. <https://doi.org/10.1145/3426428.3426924>
- [50] Marcel Taeumel. 2020. *Data-Driven Tool Construction in Exploratory Programming Environments*. Ph. D. Dissertation. University of Potsdam, Digital Engineering Faculty, Hasso Plattner Institute. <https://doi.org/10.25932/publishup-44428>
- [51] Marcel Taeumel, Jens Lincke, Patrick Rein, and Robert Hirschfeld. 2022. A Pattern Language of an Exploratory Programming Workspace. In *Design Thinking Research: Achieving Real Innovation*, Christoph Meinel and Larry Leifer (Eds.). Springer, Cham, 111–145. https://doi.org/10.1007/978-3-031-09297-8_7
- [52] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming* (San Francisco, California) (LIVE '13). IEEE Press, 31–34. <https://doi.org/10.1109/LIVE.2013.6617346>
- [53] Steven L. Tanimoto. 2023. Five Futures with AI Coding Agents. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming* (Tokyo, Japan) (Programming '23). ACM, NY, USA, 32–38. <https://doi.org/10.1145/3594671.3594685>
- [54] Christoph Thiede and Patrick Rein. 2023. *Squeak by Example* (6.0 ed.). Lulu. ISBN 978-1-4476-2948-1.
- [55] Christoph Thiede, Willy Scheibel, and Jürgen Döllner. 2024. 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* (Rome, Italy) (IVAPP '24). INSTICC, SciTePress, 661–669. <https://doi.org/10.5220/0012393900003660>
- [56] Christoph Thiede, Marcel Taeumel, and Robert Hirschfeld. 2023. 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* (Tokyo, Japan) (<Programming>). ACM, New York, NY, USA, 54–60. <https://doi.org/10.1145/3594671.3594678>
- [57] Christoph Thiede, Marcel Taeumel, and Robert Hirschfeld. 2023. 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* (Cascais, Portugal) (Onward!). ACM, New York, NY, USA, 89–102. <https://doi.org/10.1145/3622758.3622892>
- [58] V. Javier Traver. 2010. On Compiler Error Messages: What They Say and What They Mean. *Advances in Human-Computer Interaction* 2010, Article 3 (1 2010), 26 pages. <https://doi.org/10.1155/2010/602570>
- [59] David Ungar, Henry Lieberman, and Christopher Fry. 1997. Debugging and the Experience of Immediacy. *Commun. ACM* 40, 4 (apr 1997), 38–43. <https://doi.org/10.1145/248448.248457>
- [60] Ashish Vaswani et al. 2017. Attention is All You Need. In *Advances in Neural Information Processing Systems*, Isabelle Guyon et al. (Eds.), Vol. 30. Curran Associates, 11 pages. arXiv:1706.03762 [cs.CL]
- [61] Alessandro Warth, Yoshiki Ohshima, Ted Kaehler, and Alan Kay. 2011. Worlds: Controlling the Scope of Side Effects. In *ECOOP 2011 – Object-Oriented Programming*, Mira Mezini (Ed.), Vol. 6813. Springer, Berlin, Heidelberg, 179–203. https://doi.org/10.1007/978-3-642-22655-7_9
- [62] Jason Wei et al. 2022. Finetuned Language Models Are Zero-Shot Learners. arXiv:2109.01652 [cs.CL]
- [63] Jason Wei et al. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]
- [64] Jules White et al. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. 19 pages. arXiv:2302.11382 [cs.SE]
- [65] Hui Yang, Sifu Yue, and Yunzhong He. 2023. Auto-GPT for Online Decision Making: Benchmarks and Additional Opinions. arXiv:2306.02224 [cs.AI]
- [66] Wayne Xin Zhao et al. 2023. A Survey of Large Language Models. 124 pages. arXiv:2303.18223 [cs.CL]
- [67] Lianmin Zheng et al. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems*, Alice Oh et al. (Eds.), Vol. 36. Curran Associates, 46595–46623. arXiv:2306.05685 [cs.CL]

Received 2024-04-25; revised 2024-07-18; accepted 2024-08-08