Dimensions of Examples: Toward a Framework for Qualifying Examples in Programming

Toni Mattis
Hasso Plattner Institute, University of Potsdam, Germany

Lukas Böhme
Hasso Plattner Institute, University of Potsdam, Germany

Stefan Ramson
Hasso Plattner Institute, University of Potsdam, Germany

Tom Beckmann
Hasso Plattner Institute, University of Potsdam, Germany

Martin C. Rinard
Massachusetts Institute of Technology, Cambridge, MA, USA

Robert Hirschfeld
Hasso Plattner Institute, University of Potsdam, Germany

Abstract

Programming requires understanding, using, and changing abstract source code and other representations of programs. Concrete *examples* demonstrate a particular instance of their abstract behavior. Hence, they play an important role in program comprehension, specification, and testing of requirements. Authoring examples entails a range of – often implicit – decisions about the content and presentation of the example. In this work, we attempt to structure this decision space by describing a set of dimensions that characterize examples in programming. As the manual effort of creating examples is increasingly automated, e.g., through the use of generative AI, we expect this catalog of dimensions to help users and tool developers parametrize, guide, and evaluate the generation of examples in terms of the vocabulary we present here.

2012 ACM Subject Classification Software and its engineering → Software notations and tools; Software and its engineering → Documentation

Keywords and phrases Examples, Live Programming, Evaluation

Digital Object Identifier 10.4230/OASIcs.Programming.2025.13

Acknowledgements This work is supported by the HPI-MIT “Designing for Sustainability” research program. We also gratefully acknowledge the financial support of HPI’s Research School.

1 Introduction

The notations programmers use to implement programs are inherently abstract to capture the generality of a program. Examples as concrete instances of an abstract concept have been used to aid comprehension since the first documented algorithms from the Bronze Age [10], and today, their use in teaching is ubiquitous throughout maths and computer science [14]. To support programming activities, good examples play an important role in documentation [21], Q&A platforms [15], and increasingly in example-based programming environments. Currently, the rise of code-generating large language models (LLMs) drastically lowers the barrier to generating executable examples [13], while dynamic tooling helps programmers mine realistic examples from running programs [12].

This raises the question about the properties of a good example. Studies have highlighted important properties in general settings like documentation and Q&A platforms [15] that mostly focus on the context surrounding the exemplary code, e.g., stepwise explanation, links to further resources, alternative solutions, or the description of limitations. Most importantly, an example should exist at all [21].

In this work, we will narrow down the definition of an example, focusing on the concrete actions (e.g., the code) and data (e.g., parameters) that demonstrate the use or implementation of a part of the system at run-time. This covers the example concept from example-based programming environments, documentation, and unit tests. We asked ourselves what type of example we would like an LLM to generate, in which ways hand-crafted, generated, or mined examples differ, which "dimensions" we want to control as programmers, and which dimensions our tooling supports.

Example

As an example, let's say we need to understand the behavior of the `sorted` function in Python and are presented with the examples below. The aspect that changes from example 1 to example 2 is the length of the sorted list, introducing complexity. Longer lists do not necessarily convey more information. Still, if we shorten it too much (as in Example 3), it might already illustrate more of an edge case rather than a representative situation. To better explain the function in the context of an exemplary web app, example 4 introduces domain-specific objects rather than generic integers, and independently increases heterogeneity without adding more list items than example 1. Example 5 replaces the "foo" users with more realistic users, demonstrating how the timestamps were intended to look (and coincidentally also revealing that users are identified by email address). A comment can add more explanation. We see that we can vary different aspects of the example almost independently, and many have a context-dependent sweet spot where being too far on one end of the axis diminishes utility.

Listing 1 Varying an example for Python's sorted function along different dimensions.

```
Example 1
print(sorted([3, 1, 2]))

Example 2
 → higher complexity
print(sorted([10, 6, 3, 4, -1, 7, 3, 3]))

Example 3
 → lower complexity, higher exceptionality
print(sorted([]))

Example 4
 → wider scope, closer to domain
from model import User
print(sorted([
     User('foo', created_at='2025'),
     User('bar', created_at='2023')
     User('baz', created_at='2024')],
     key=lambda user: user.created_at))

Example 5 (like 4, but with list items like those below)
 → closer to domain, more explanation
... User('alice@example.com', created_at='2025-06-02T10:00:00Z'), ...
# We can sort ISO 8601 timestamps lexicographically
```

Table 1 Overview of the dimensions of examples.

Complexity	Explanation
Exceptionality	Executability
Closeness to Domain	Secondary Notation
Scope	Interactivity
Abstractness	

Based on our experience building and using example-based programming environments, we propose a set of dimensions inspired by the *Cognitive Dimensions of Notations (CDN)* [2] and the *Technical Dimensions of Programming Systems (TDPS)* [9] frameworks.

We anticipate that such a framework can be helpful for comparing and evaluating examples, e.g., by designing questions for user studies that target these dimensions specifically. Moreover, these dimensions can parametrize the generation of examples, e.g., by specifying whether the example should be high or low in certain aspects, or giving feedback that the next generation attempt should be higher/lower than a previously generated example. Dimensions that certain tools cannot control give rise to exploring future tooling.

2 Examples in Programming

2.1 What is an Example?

In the scope of this work, an example is considered an instance of a set of behaviors of a system or parts thereof. To explain its subject, an example needs to establish some computational context, which can include actions (manual or automated) and data. We consider these actions and data as integral parts of the example, e.g., we can illustrate how an API should be used by showing how it is invoked (action) with example parameters (concrete data).

2.1.1 Roles of Examples

When we consider programming to consist of several interleaving and repeating activities, examples play different roles in different activities and have different lifetimes. From our example-focused perspective, we consider three types of activities:

Exploration. Activities to better understand or recall how to use data, tools, and abstractions to approach a programming task. In exploratory phases, programmers benefit from reading examples in documentation, on Q&A platforms, from advice given directly by peers, and increasingly from generative AI chats. During the activity, they might increasingly come up with their own ad-hoc examples to try something out or explore a problem in a debugger using a run-time example. Hence, exploration narrows down the lifetime of examples, as initial knowledge might be derived from long-lived media, while the eventual verification of whether newly written code performs as expected might happen with a non-persistent ad-hoc input.

Evaluation. Activities where programmers put a (step towards a) solution to test. In contrast to exploratory activities, where examples tend to increase programmers' understanding, the role of examples now shifts toward providing feedback and verifying existing behavior. Typical examples are those found in unit tests or benchmark data. Programmers might come up with their own inputs they like to test or resort to those specified in the requirements. Edge cases can become more prominent.

Explanation. Activities that accompany the dissemination of programs and knowledge about them. Programmers might write or edit examples in comments, tests, or documentation that should serve a wider audience and serve as input for exploration activities.

We understand these activities to be recursive, as exploratory activities often entail a rapid succession of smaller exploration and evaluation steps, possibly discarding experiments before moving on.

We argue that knowing which activity an example should support is crucial to understanding not only its scope but also the tooling involved.

2.2 The Role of Tools

Tools play an important role in how examples can be created and consumed. The traditional tools programmers use to edit and view code and documentation already provide a basic platform that allows them to edit examples as if they were code or documentation, e.g., as code comments in source files or code blocks in text documents. Such examples are not reified, hence they are easily overlooked, can be incorrect or outdated, and require manual upkeep and execution. However, they tend to benefit from text-based tooling, such as version control and full-text search.

2.2.1 Tests as Syntactic Examples

Although primarily geared toward verifying program behavior, unit tests are a code-level mechanism to specify examples, name them, and make them executable through a test runner. When writing unit tests, programmers can benefit from existing features of their programming environment, such as syntax and error highlighting, code completion, or debuggers. To better modularize test logic, many testing frameworks offer constructs to implement fixtures, which are reusable and reproducible example instances of certain system components. The language PyRet[1] even has syntactic support to specify exemplary tests using a `where`-clause after a function definition. Examples specified in tests are required to be executable, which sets a lower boundary on their complexity (e.g., requiring authors to substitute parameters through mock objects when they cannot be left out entirely) and frequently cover edge cases that are not necessarily representative but can still serve an important illustrative function.

2.2.2 Example-based Live Programming

The use of examples in programming motivates the design of environments that specifically support examples beyond a program's regular code. Several designs have been proposed that treat examples as first-class entities in the programming environment. These examples can then be used as entry points to provide fine-grained dynamic data that helps explain and evolve programs by example. A few examples of such systems are described below:

Exemplars. In the *Newspeak* programming language, methods can be accompanied by example parameters and class instances (*Exemplars* [3]) that enable programmers to always have a run-time instance of that method available.

Example-centric Programming. By tracing the execution of code examples through the code they illustrate, Example-centric Programming displays the evolution of concrete data next to its relevant code [5].

[1] `https://pyret.org/` (last accessed 2025-05-05)

Babylonian Programming. Babylonian Programming [18] introduces editor elements to specify and select example instances and parameters for a section of source code (e.g., a method) and provides *probes* – an in-situ editor feature – that display dynamic data yielded by executing the active examples at the code location where they occur. This mechanism works across module boundaries. It can circumvent side effects and simulate user inputs using *replacements* that insert an exemplary return value for a selected expression. Implementations have been presented for Squeak/Smalltalk [19], JavaScript in Lively4 [18], the polyglot GraalVM runtime in VSCode [16], and the Godot game engine [11].

Kanon. Kanon [17] is a domain-specific example-based live programming environment for data structure manipulation. The consequences of each code statement are explained graphically.

SnapPy. The SnapPy [6] tool for Python takes example-based programming one step further and not only presents the current state of an example at run-time but also allows programmers to demonstrate by example what should happen with the current state, prompting the system to synthesize new code. This is an instance of *programming by example* [8].

With the introduction of example-aware UI components, programmers no longer need to care about where and how to implement them at the code level and gain the capability to view dynamic data within their code editor. Editor components that are not restricted to text can use rich visualizations, e.g., plotting a value over time rather than printing it as a number, viewing the content of a drawing buffer, or presenting structured data as tables or collapsible trees.

With closer proximity to code and fine-grained entry points near the code of interest, we gain the capability to provide more immediate feedback on every code change, as we can scope execution to just the relevant part rather than the program's main entry point. Correspondingly, tools can support more experimentation.

2.3 Generating Examples

Example creation entails manual effort, which limits their availability to parts of the system where programmers have already invested this effort. Several approaches can automate example generation:

Example mining. Example mining utilizes run-time data to obtain realistic objects during test runs, user interaction, or debugging sessions that can then serve as examples in an example-based programming environment [12]. Subsequent (automated) pruning might be necessary depending on the use case, for example, to reduce the size of collected objects for storage. In other instances, however, a complete snapshot of program state can be helpful, e.g., in interactive media like simulations and games [11].

Fuzzing. Using fuzzing, we can systematically construct a large number of inputs to discover unintended behavior, yielding examples that serve as a starting point for debugging. Similar to Example-based Live Programming, which generates immediate feedback using a realistic example, fuzzers can generate edge-cases for code under development [7][2].

Unit Test Generation. Unit test generation focuses on deriving tests from existing code and increasing coverage. Currently, generative AI is frequently used to synthesize the test code, as it can match identifiers and other patterns to approximate the original intent and synthesize representative test data [4].

[2] Review note: This statement depends on a work that is concurrently under review.

Example Generation. Example generation specifically targets the generation of executable examples. In contrast to tests, no assertions or verification logic needs to be generated as the focus is on representative example data rather than verification. Similar to test generation, generative AI is a promising approach [13].

Especially in settings where examples are generated, programmers might want to be in control of certain qualities of the example, e.g., how much of a mined example might be pruned and how complex an object structure a generative method should construct. In this work, we make the dimensions along which we aim to exert control over example generation explicit.

3 Dimensions of Examples

In the following section, we describe our proposed set of dimensions that characterize an example. We envision them to serve as terminology to communicate how pronounced each dimension is in an example and as parameters that future example-generation tools can control. These dimensions are not fully independent from each other, and revising an example along one dimension can positively or negatively influence other dimensions. We distinguish between dimensions related to the content an example entails and those that describe the presentation of the example within its tooling context. We subsequently demonstrate the dimensions using three instances of examples.

3.1 Content-focused Dimensions

This set of dimensions describes the attributes of the examples' contents.

3.1.1 Complexity

We define complexity as the amount of data or actions that the example contains. Simple ways to increase complexity are including more items in a collection, constructing a deeper graph or tree structure, or using longer strings. Often, the least complex example for functionality is an edge case and not necessarily representative (e.g., the empty list), while adding complexity has diminishing returns (e.g., sorting a list of 15 items is rarely more insightful than 10 items), which implies the existence of a sweet spot that can change per use case.

3.1.2 Exceptionality

Exceptionality corresponds to how far outside the expected or typical distribution an example is situated. Examples that rank highly in this dimension will often be called edge cases. They can be useful to understand how the system handles error conditions, e.g., by providing invalid or incomplete inputs or constructing a rarely anticipated precondition like missing file permissions. The more unlikely or unimportant the example for the considered domain, the higher its exceptionality.

What constitutes exceptionality depends on context: the empty list is more exceptional when computing the arithmetic mean, and less exceptional when used as the default for an optional parameter in an API. In the security domain, what looks like an exceptional example might as well illustrate a typical attack vector that the implementation must mitigate.

This dimension is connected to complexity, as examples that are too simple or too complex tend to be exceptions.

3.1.3 Closeness to Domain

An example can explain the same logic using meaningless inputs or data that is more representative of the domain. Examples include using familiar names of people, places, products, etc., to identify domain objects compared to using "foo" and "bar" or "Lorem Ipsum". In situations where some logic becomes part of the larger system, e.g., a sorting algorithm being used in a calendar app, sorting a list of integers might still explain the sorting routine in isolation, but sorting a list of events by closest date would illustrate its use in the context of the domain.

Whether closeness to the domain is better depends on the context in which the example is used: for code that parses a user's name, using "First Last" as the example string can help during debugging to see if first and last names have indeed been parsed correctly into their respective variables. In this case, the example would be closer to the implementation than the domain. On the other hand, not using realistic examples can cause bias, e.g., programmers not expecting non-Latin characters in a name or neglecting middle names.

3.1.4 Scope

Scope complements the complexity dimension by describing to what degree the example connects and integrates multiple aspects of the illustrated concept. For example, a plotting library could illustrate in a single example the use of different colors, line styles, markers, logarithmic axes, and labels[3]. While this often increases complexity, complexity itself can be increased without affecting scope by just adding more data without varying plot parameters.

Demonstrating the interplay between libraries, adding context that might be hard to deduce for novices, and explicitly specifying optional or default parameters can all add to scope. In this way, scope determines which related aspects beyond the example's core are included.

The opposite would be reductionist examples that explore a single aspect in isolation. To understand multiple aspects, multiple examples would be needed. Although this often results in simpler examples, dependencies and interactions between parameters would still remain abstract.

3.1.5 Abstractness

Making examples executable often requires them to be as concrete as possible, down to instantiating primitive data types.

When executability (see Subsubsection 3.2.2) is not a concern, examples can gradually abstract away irrelevant details using symbolic placeholders (that the user needs to fill in before execution). For example, for parsing a unified diff we might explain the header of a change as `@@ -2,3 +2,4 @@`, but also as `@@ -a,b +c,d @@` or `@@ -range_removed +range_added @@`, increasing the abstractness of our example string. Depending on the parsing logic, we might even succeed in executing the more abstract examples up to a point.

Concrete does not necessarily mean primitive data as in strings or numbers: One could also explain *monads* using the example of the *list* type – in this case, concrete "data" would actually mean *functions* (the function to wrap an element in a single-item list and the `flatmap` function constitute concrete versions of the unit and bind operators).

[3] For concrete examples, see the Matplotlib documentation at `https://matplotlib.org/stable/gallery/index.html` (last accessed 2025-03-21)

In testing, more abstract examples serve as inputs for property-based testing or concolic execution [20]. In such settings, symbolic parts of an example can still be considered during execution.

3.2 Presentation-focused Dimensions

The following set of dimensions is highly dependent on the tools through which examples are being authored and used.

3.2.1 Explanation

This dimension is the degree to which the example relies on non-code explanation. Depending on whether the example is viewed as part of external documentation, code, or within an example-based programming environment this might take the shape of (rich) text surrounding the example, code comments, or the use of extra features introduced by tooling (e.g., a descriptive name as in some Babylonian programming environments).

3.2.2 Executability

Executability corresponds to how little manual effort stands between seeing the example and observing its effects. In test suites, notebook-style programming, and example-based programming environments, examples are usually executable by construction. Examples more removed from the programming environment might be one copy-and-paste action away from running or might need adaptation or preparation (e.g., when dependencies or preconditions are not made explicit in the example or are only stated textually).

We observe that tools can greatly increase the executability of examples by hiding the effort of setting up the context in which the example can run: While writing a unit test often requires a separate module or file, importing the unit under test, defining a test case class or method, writing setup code, and invoking the test runner, an editor supporting Exemplars or Babylonian programming can provide a user interface where any parameter can be directly set to an example value right where the method is defined and execution results are automatically obtained by running the code on every change.

3.2.3 Secondary Notation

In the cognitive dimensions of notations framework [2], this dimension describes the capability of a notation to vary and convey information without changing the semantics of a program, such as positioning of elements, highlighting, or formatting. Although currently underused in example-based programming environments, such features promise to make examples more comprehensible. For example, letting users color a part of an example could allow them to convey importance, but also allow dynamic tooling to reuse this color to highlight outputs related to that part of the example.

3.2.4 Interactivity

This dimension maps the degree to which the example can be “experienced” through on-the-fly changes. Explorable Explanations [22] are a proposal that encourages *active reading* by experimentation with examples included in a document. Kanon [17] supports domain-specific visualizations of data structures that can be edited. While these interactive representations are often domain-specific, recent developments aim to support general-purpose live objects. For example, Exploriants [1] supports interactive UIs as examples, including a playable game.

3.3 Three Instances

In the following, we discuss three instances of examples using the framework.

Matplotlib includes a wealth of examples in its documentation. Often, these are coupled with the visual output of executing the code. Correspondingly, these examples maintain a large *scope* – even though they are designed to illustrate only one aspect of the API, such as legends, the examples include setup and teardown, as well as other components that may influence the appearance of legends. These examples are *executable* as-is. The *complexity* of the examples varies: for instance, some examples make use of datasets that are bundled in the library. The complexity of loading and deserializing data is thus hidden and what remains is a very concrete, as opposed to *abstract*, example. Using a realistic dataset also emphasizes *closeness to the domain*. Notably, even if the example is only meant to illustrate how a method is correctly invoked, the documentation tends to choose data as it would likely appear in real programs. To keep the examples focused, they tend to reduce exceptionality and may instead favor listing exceptional cases through prose. The degree of *explanation* tends to be high – both the documentation text and code comments add further insights.

In Squeak/Smalltalk, it is common to provide commented-out lines of code at the start of a method that execute an example, often the same method. These examples unfold their illustrative value often only through *interactivity*. Either the user experiments with the output to understand the impact of the parameters, or the user invokes the debugger on the example snippet and steps through relevant invocations to understand the flow and nature of data. Again, these examples maintain high *scope*, *concreteness*, and *executability*. Here, too, exceptionality is likely to be low, as the snippets are often the archetypical invocation of the component. These snippets are typically provided as-is, with a low degree of *explanation*, sometimes giving context on the expected output.

To take a shortcut during prototyping, programmers may choose to defer writing tests and instead include instances of input to a given algorithm as a comment. Here, the examples have low *executability*, depending on how *abstract* they were formulated. An abstract formulation may include characteristics of inputs ("users with an unread notification"), while a concrete formulation may include pieces of (pseudo) code that produce the required values in the host language, ready to be copied into an invocation. The *scope* of such examples will typically be rather narrow, only demonstrating expected values. Quite often, *exceptionality* could be high, as programmers add example inputs they initially forgot to handle. The degree of *explanation* will vary. Especially for complex use cases, where programmers choose more abstract examples, they may also choose to provide additional context in the form of explanations.

4 Outlook and Conclusion

We hope that these dimensions can raise awareness along which axes examples can vary. Especially the content-focused dimensions might serve as a starting point for developers of example-based tooling to explicitly support them as parameters that users control. This can range from implementing them as sorting or filtering criteria over feedback mechanisms (e.g., "make the example more/less complex") to better training, fine-tuning, or prompting LLMs to respond to the user's specification in terms of these dimensions.

While we expect the content-focused dimensions to parametrize examples, we see the presentation-focused dimensions as those that give authors more creative freedom over their example where their tooling allows it. This motivates the exploration of tools that allow for explanations or secondary notation, means to interact with an example, or to make it easier to see the example run with minimal effort.

While there are likely more dimensions in which two examples can differ, we reduced our model to seven key dimensions that were most pronounced when comparing different tools. The next step is testing and refining this model using different sources of examples.

References

1 Tom Beckmann, Joana Bergsiek, Eva Krebs, Toni Mattis, Stefan Ramson, Martin C. Rinard, and Robert Hirschfeld. Probing the Design Space: Parallel Versions for Exploratory Programming. *The Art, Science, and Engineering of Programming*, 10(1):5:1–5:33, February 2025. doi:10.22152/programming-journal.org/2025/10/5.

2 A. F. Blackwell, C. Britton, A. Cox, T. R. G. Green, C. Gurr, G. Kadoda, M. S. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, and R. M. Young. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In Meurig Beynon, Chrystopher L. Nehaniv, and Kerstin Dautenhahn, editors, *Cognitive Technology: Instruments of Mind*, Lecture Notes in Computer Science, pages 325–341. Springer Berlin Heidelberg, 2001.

3 Gilad Bracha. Enhancing liveness with exemplars in the newspeak ide. https://newspeaklanguage.org/pubs/newspeak-exemplars.pdf. Accessed: 2025-02-04.

4 Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 572–576, Porto de Galinhas Brazil, July 2024. ACM. doi:10.1145/3663529.3663801.

5 Jonathan Edwards. Example centric programming. *SIGPLAN Not.*, 39(12):84–91, December 2004. doi:10.1145/1052883.1052894.

6 Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. Small-Step Live Programming by Example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, pages 614–626, New York, NY, USA, October 2020. Association for Computing Machinery. doi:10.1145/3379337.3415869.

7 Marcel Garus, Jens Lincke, and Robert Hirschfeld. Fuzzing as editor feedback. In *Proceedings of the Programming Experience 2025 (PX/25) Workshop*, Programming '25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (OASIcs), 2025. doi:10.4230/OASIcs.Programming.2025.18.

8 Daniel Conrad Halbert. *Programming by example.* PhD thesis, Department of Electrical Engineering and Computer Sciences, Computer Science Division, University of California, Berkeley, 1984. AAI8512843.

9 Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. Technical Dimensions of Programming Systems. *The Art, Science, and Engineering of Programming*, 7(3):13:1–13:59, February 2023. doi:10.22152/programming-journal.org/2023/7/13.

10 Donald E. Knuth. Ancient Babylonian algorithms. *Communications of the ACM*, 15(7):671–677, July 1972. doi:10.1145/361454.361514.

11 Eva Krebs, Toni Mattis, Marius Dörbandt, Oliver Schulz, Martin C. Rinard, and Robert Hirschfeld. Implementing Babylonian/G by Putting Examples into Game Contexts. In *Proceedings of the Programming Experience 2024 (PX/24) Workshop*, Programming '24, pages 68–72, New York, NY, USA, July 2024. Association for Computing Machinery. doi:10.1145/3660829.3660847.

12 Eva Krebs, Patrick Rein, and Robert Hirschfeld. Example Mining: Assisting Example Creation to Enhance Code Comprehension. In *Proceedings of the Programming Experience 2022 (PX/22) Workshop*, Programming '22, pages 60–66, New York, NY, USA, December 2022. Association for Computing Machinery. doi:10.1145/3532512.3535226.

13 Toni Mattis, Eva Krebs, Martin C. Rinard, and Robert Hirschfeld. Examples out of Thin Air: AI-Generated Dynamic Context to Assist Program Comprehension by Example. In *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming*, Programming '24, pages 99–107, New York, NY, USA, July 2024. Association for Computing Machinery. doi:10.1145/3660829.3660845.

14 Kasia Muldner, Jay Jennings, and Veronica Chiarelli. A Review of Worked Examples in Programming Activities. *ACM Transactions on Computing Education*, 23(1):13:1–13:35, December 2022. doi:10.1145/3560266.

15 Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example?: A study of programming Q&A in StackOverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34, September 2012. doi:10.1109/ICSM.2012.6405249.

16 Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. Example-based live programming for everyone: Building language-agnostic tools for live programming with LSP and GraalVM. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2020, pages 1–17, New York, NY, USA, November 2020. Association for Computing Machinery. doi:10.1145/3426428.3426919.

17 Akio Oka, Hidehiko Masuhara, and Tomoyuki Aotani. Live, synchronized, and mental map preserving visualization for data structure programming. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2018, pages 72–87, New York, NY, USA, October 2018. Association for Computing Machinery. doi:10.1145/3276954.3276962.

18 David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. Babylonian-style Programming. *The Art, Science, and Engineering of Programming*, 3(3):9:1–9:39, February 2019. doi:10.22152/programming-journal.org/2019/3/9.

19 Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, Fabio Niephaus, and Robert Hirschfeld. Implementing Babylonian/S by Putting Examples Into Contexts: Tracing Instrumentation for Example-based Live Programming as a Use Case for Context-oriented Programming. In *Proceedings of the 11th ACM International Workshop on Context-Oriented Programming*, COP '19, pages 17–23, New York, NY, USA, July 2019. Association for Computing Machinery. doi:10.1145/3340671.3343358.

20 Koushik Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 571–572, New York, NY, USA, November 2007. Association for Computing Machinery. doi:10.1145/1321631.1321746.

21 Henry Tang and Sarah Nadi. Evaluating Software Documentation Quality. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 67–78, May 2023. doi:10.1109/MSR59073.2023.00023.

22 Bret Victor. Explorable explanations. https://worrydream.com/ExplorableExplanations, 2011. Accessed: 2025-03-21.