Example Mining

Assisting Example Creation to Enhance Code Comprehension

Eva Krebs
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
eva.krebs@hpi.uni-potsdam.de

Patrick Rein
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

ABSTRACT

Programmers often use examples with concrete values to better understand code. Code by itself is abstract, which empowers it to be used for a variety of uses, but can be difficult to grasp by developers. Babylonian Programming addresses this by allowing programmers to concretize their code by defining and visualizing examples directly in the code itself while editing.

Currently, Babylonian Programming implementations such as Babylonian/S require programmers to define examples manually. For examples containing small objects this is straightforward, however when creating large or complex objects it is not. Sometimes, workarounds such as copying existing code pieces or trying to recreate existing objects are used to reuse information already present in the system, but these workarounds can be error-prone and also time-consuming.

In this paper, we propose Example Mining to address this issue by providing techniques and integrated tools to mine examples from existing sources similar to concepts from run-time tracing and test case extraction. Example mining introduces concepts to mine examples from tests, debugging sessions, and traces of actual usage of the system under development. All tools were implemented for Babylonian/S in Squeak/Smalltalk.

We demonstrate the usefulness of the tools through walkthroughs. Our tools provide examples for many methods immediately and automatically. As a consequence, programmers have more immediate access to dynamic feedback on their abstract code, making code comprehension and debugging more effective.

CCS CONCEPTS

• **Software and its engineering → Development frameworks and environments**.

KEYWORDS

live programming, exploratory programming, example-based programming, babylonian programming, examples, squeak, smalltalk

‹Programming› '22 Companion, March 21–25, 2022, Porto, Portugal

ACM ISBN 978-1-4503-9656-1/22/03…$15.00
https://doi.org/10.1145/3532512.3535226

ACM Reference Format:
Eva Krebs, Patrick Rein, and Robert Hirschfeld. 2022. Example Mining: Assisting Example Creation to Enhance Code Comprehension. In *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming (‹Programming› '22 Companion), March 21–25, 2022, Porto, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3532512.3535226

1 INTRODUCTION

Code itself is abstract, allowing it to have a wide variety of uses, but also making it potentially difficult to understand. To combat this, programmers may run the code in their mind and assign concrete values to the abstract rules. Having concrete values can enhance program comprehension, especially if developers see a specific piece of code for the first time [16, 34]. But there are several other situations where a programmer might want to use examples with concrete values. For instance, to document how code should behave, programmers might write tests or include small scripts in the documentation.

However, creating new examples quickly on-demand can be difficult. The user will have to create appropriate objects from scratch and set all necessary attributes for both the receiver of the method, as well as any parameters. This can be simple for methods that do not require complex objects. However, the required objects might consist of many attributes, have a deep object structure, or be otherwise complex. Configuring such an object, even if it is clear what is needed, can already take developers a long time and be prone to small mistakes. But often it is not trivial which parts of a complex object are necessary and what values these attributes should have, making the process even more time-intensive and error-prone.

Currently, the manual creation of examples results in programmers creating workarounds to reuse data from existing scripts or traces of previous method invocations. Programmers may have to search to find appropriate tests or documentation. Set-up code from tests could be copied to configure suitable objects. References to or copies of objects used in the system might be created. These workarounds are usually time-intensive, error-prone, and might lead to unnecessary code duplication.

Integrated tools could improve the creation of examples for programmers. Statically available sources, such as tests, could directly be used as an example instead of duplicating code pieces or objects of the tests. Dynamic information and existing objects could be recorded and added as examples without the error-prone process of trying to add them with workarounds. Often static analysis is preferred to dynamic analysis, but creating reliable tools that support dynamic analysis and adding information from it as examples

could increase its use by developers [35]. All in all, there are multiple sources that could potentially be useful as examples, but are currently impractical to use.

We propose Example Mining to assist programmers with example creation. Example Mining includes techniques to mine examples from tests, debugging sessions, and traces of actual usage of the system under development. We created dedicated tools to mine examples and present potential workflows.

In this paper we will first introduce example-based programming systems as well as the related field of live programming in section 2. The concept of Example Mining will be described in section 3 followed by potential use cases in section 4. The paper ends with a discussion of related work in section 5 as well as a conclusion and future work in section 6.

2 LIVE PROGRAMMING AND EXAMPLES

There are several programming concepts with a focus on using examples to enhance program comprehension. This section will introduce the general concept of live programming and example-based live programming. As our approach is based on the Babylonian Programming environment, we will introduce it in detail including three of its implementations.

2.1 Example-Based Live Programming

Live programming aims to provide immediate feedback to enhance explorability and comprehensibility [9, 28]. To provide this feedback, editing and running code is combined to achieve faster feedback cycles. There are several ways in which an IDE can support liveness, one being user-made edits triggering updates [31, 32]. The ways in which the feedback, program traces or results, can be explored is also a significant part of live programming [21].

Example-based live programming adds concrete examples to abstract code, for instance in the form of example invocations of a method. For one, programmers can use these concrete invocations to get a better grasp of the role of parameters when reading a method. Second, with the help of these examples, example-based environments can display live feedback on the behavior of the method. For instance, the Example Centric Programming editor displays a full run-time trace next to the code that is updated whenever code is changed [4]. The various, existing example-base live programming environments differ, among other aspects, in the degree to which examples are made explicit, the kind of feedback that is provided, and the support for feedback across modules [4, 11, 14, 26].

2.2 Babylonian Programming

Babylonian-style Programming, or Babylonian Programming, is an approach that aims to display examples and run-time data directly within the source code. The example encompasses the data needed to run code that invokes the method under development and is defined by the programmers. Probes can then be placed on arbitrary expressions to see what the expression evaluates to during the execution of an example [21]. Multiple implementations of Babylonian Programming exist that share certain concepts, such as placing probes to visualize a code piece or defining examples.

Babylonian/S. Babylonian/S is the main implementation of Babylonian-style Programming [27], implemented in Squeak/Smalltalk [7, 12].

The main concepts are defining examples, which provide some sort of code execution from which concrete traces of data can be gained, and adding widgets to code, allowing data to be visualized and other interactions with the code. An example can be named, can be set to whether it is inactive or active, and can contain information for all objects that are needed to run the method. Programmers can define the receiver object and the parameter objects by either writing small scripts or referring to class methods. Whenever the method source code is changed, all active examples are run.

If at least one example is defined and set to active, Babylonian widgets in turn are activated and display according information. The widgets are added by developers to parts of the code that they are interested in. The most common widget is a probe, a widget that visualizes the data of where it is placed, but there are other widgets such as sliders or replacements that can be used to change values of the code at run-time.

There are multiple kinds of examples in Babylonian/S. All example types can have a set-up and a tear-down script that are executed before and after the example is run respectively. Similar to tests, this can be used to configure needed state. One type of examples are *script examples*. These examples rely on a user-written script that invokes the target method. The script itself can directly include the method invocation, but it is also possible to invoke a different method that indirectly uses the target method. If the target method is not called, the probes and other widgets remain empty. As scripts are versatile, this example type allows developers the most freedom during example definition. A script example is especially useful if developers know how to invoke the target method indirectly but not how to invoke it directly.

Other Implementations. The first prototype of Babylonian-style Programming was implemented in JavaScript in the live programming environment Lively4 [17, 26]. Babylonian/JS introduced the main concepts of Babylonian Programming. Babylonian Programming is also a part of an example-based live programming plug-in for Visual Studio [23]. The plug-in supports polyglot programming, which means that using and combining multiple programming languages is supported. To support more than one language, core Babylonian features were implemented in a language-agnostic way.

3 EXAMPLE MINING

There are several potential sources to mine examples from. We contribute concepts and workflows to mine examples from three selected sources. This section will introduce the three tools. The tools were originally designed for Babylonian/S2.2 based on concepts applicable to several example-based environments.

In Babylonian/S, an example is defined for a method. A method that a user wants to add an example to will be called *target method*. A visualization of an example in Babylonian/S can be seen in Figure 1.

3.1 Overview

Sources for examples already exist in the system and can be accessed, but not easily integrated as Babylonian examples. Given a target method, programmers could search for a fitting test in the test packages. Or the programmers could write a small script that could be debugged or otherwise inspected. Adding, and in some

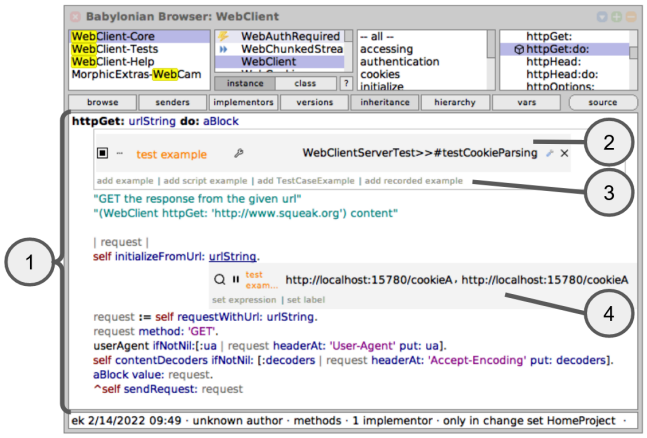

Figure 1: A Babylonian/S code browser displaying a method (1). At the top of the method examples can be defined and turned on/off. In this case one example was created and is active (2), more could be created using the button bar below the existing example (3). The probe (4) is placed in the code to visualize a property.

cases finding, these potential examples however would without tools only be possible through workarounds.

This paper will introduce three tools to add, and if needed enhance the finding process of, examples:

- A test-based Example Mining tool, which enables programmers to find relevant tests and directly add a chosen test as an example
- An addition to debugging, which can add an example based on the current state in the debugger
- An example recording tool, that can record all traces of method calls of a target method and then to add a selected recording as an example

While tests are a static and explicitly documenting source of examples, examples mined from debugging and recording arise from dynamic processes triggered by user actions. This will effect both how sources for examples can be found as well as their later representation as examples.

3.2 From Tests to Examples

Tests are an integral part of software development that could potentially be a vast source for examples. Testing not only checks for correct behavior and results, it also documents purpose, usage, and behavior of the code. Size and intent of tests varies. An integration test might span multiple modules and execute many methods, but a lot of them will not be invoked directly in the test method but be called indirectly. A unit test meant to test a specific method by contrast will likely invoke said method directly. In both cases running the tests could provide interesting data for method invocations and thus the tests could make good examples, but the way to determine whether the tests are of interest to a given method may vary.

To support the creation of test-based examples for a target method, this paper proposes a tool that can be opened for the target method and has both means to find relevant tests as well as to add a selected test as an example. The search for relevant tests should be configurable, as which tests are the most relevant might depend on the intent of the developer. For methods that are used very often, especially indirectly, in the system programmers might prefer to only see tests that directly invoke the target method or if possible are from a dedicated test class for the class of the target method. For other methods that are rarely used and do not have dedicated tests, any invocation from a test might be interesting, even if it is indirect and if all tests have to be searched to find it. If many results are found, applying more criteria might not only yield more relevant tests but also less yet more focused tests that are easier for the programmer to look through and less overwhelming. Once a developer has found a promising test, or multiple, that test should be able to be added as an example.

To create a test-based example, it likely be best to introduce a new kind of example whose purpose is to run a test. Existing example types could used to run a test in an example, a script example for instance. The script example could have a script containing either code to run the test or a copy of the test code. Copying code or objects from tests however might be problematic, for example if the test is refactored a user might prefer the refactored version, but the scripts will still contain the old code. Having a script that runs the example would solve that, but might also not be optimal long-term. If the way of running tests changes, all scripts would need to be changed, while a test example could encapsulate that behavior so it only has to be changed once. Also, the scripts would have no connection to the test, the meta-information about the example being based on a test would be lost. Knowing that an examples is based on a test creates more possibilities, both in how the example is presented as well as further features that only work with test methods.

This paper proposes a *test example*, that can run a test method and persists the needed information. In most systems, persisting the name of the test class as well as the selector of the test method make it possible to identify the test method. The test can then be run with means provided by the test framework, which might for example include automatic set-up and clean-up functionality.

3.3 From Debugging to Examples

A debugger allows further insights into method executions. In Squeak/Smalltalk, the debugger allows access to the entire method stack and to step through methods individually. All needed information, such as receiver and arguments of the currently debugged method, is accessible and can be explored by the programmer.

It is possible for users to debug a specific piece of code on purpose, but debuggers are also automatically opened on errors and breakpoints. This means the state visible in the debugger can be interesting for several reasons. If the debugger was used to explore, maybe an interesting method execution was found by chance. Maybe a piece of code was debugged with the intent of seeing concrete values for a specific method. A debugger opened for an error might contain an invocation that is interesting as an illustration long-term. Or instead of investigating the error further in the debugger, the programmer wants to use other available tools for exploration.

If a debugger is deemed to contain an interesting method invocation by the developer, it should be possible to use that invocation as an example. Whether the content of the debugger makes a good example can be decided spontaneously as the only tool needed should be the debugger itself. This makes it possible for developers to explore the system and create examples easily at each point.

The information provided by the debugger contains the currently explored method, the method's receiver, and the method's arguments. Since receiver and arguments are directly available, they can be used to create a method example, an example type that needs receiver and arguments to be defined individually. This means we can save examples from the debugger as method examples. The example should not be created with the original objects from the debugger but with copies, as the objects in the debugger might still be in use. If the objects are still in use, the example working with them could alter other behavior in the system. Similarly, if the used objects are changed outside the example that would impact future example runs. Also, the objects in the example should be saved in a way that ensures that if the example is executed several times the receiver and argument objects do not change. This means side-effects from the method should not be persisted, since the original receiver and arguments state from the debugger created the interesting execution. If the objects are altered, the method execution and values used in illustrations might change.

3.4 From Traces to Examples

The environment by itself and interactions with it already lead to several method executions. The data in these could in some cases also make useful examples. A way to access all traces of a target method could be helpful if the developer knows when the method is called and wants a specific trace. It could also help to see if a method is triggered at all and with what data.

There are several ways in which useful traces could be created. The system might do some actions in the background, like checking for new emails. The programmer might interact with a tool or other application that internally uses the method of interest. Sometimes it might be interesting what traces are generated by a specific script.

To access these traces, this paper proposes a tool that can record traces for a target method and add examples based on traces. The developer can start and end recording to collect traces in a user-defined period of time. While recording, all actions taken by the developer or the system itself work as normal, but for all invocations of the target method receiver and arguments are recorded. Recording only the target method means that this tool can be used to reduce scripts that could also be used in script examples to only the method invocation needed for the example. All other, potentially time- or resource-intensive, parts of the scripts can be removed by omission.

A recorded trace consist of receiver and arguments for a target method. Similarly to the handling of the data found during debugging, these objects can be used to create method examples.

Recording Traces. To record traces relevant to the target method, all invocations of that method should be able to be recorded on demand. There are several ways to record method invocations, from vm-level implementations to recording tools. For this paper, a recorder that is already provided by Babylonian/S will be used. Like Babylonian/S itself, the recorder is based on code instrumentation. At the beginning of the method, the recorder adds a hook to itself that saves a copy of the receiver and arguments of the method as recording.

4 EXAMPLE MINING IN ACTION

In the following, three scenarios will be introduced in which examples can be mined with the created tools. All scenarios will center on the WebClient, a class that supports sending web requests and related functionality. In some cases, tools using the WebClient might be featured.

4.1 From Tests to Examples

A programmer might want to add an example to the #httpGet:do: method of the WebClient. It might be unclear what parts of a WebClient instance need to be configured for the method to run without error, but as an integral method it is likely to be tested.

To add a test-based example, first the target method, #httpGet:do:, needs to be opened with a code browser. Then, the tool for adding test-based examples needs to be opened for the method. The tool can be seen in Figure 2. Immediately, results based on the default search configuration will be displayed. If the user wants specific criteria to be fulfilled, for example that the target method is definitely invoked by the test, the user can change which conditions are active and trigger the search again. A fitting test, e.g. #testCookieParsing, can then be added as an example to the method.

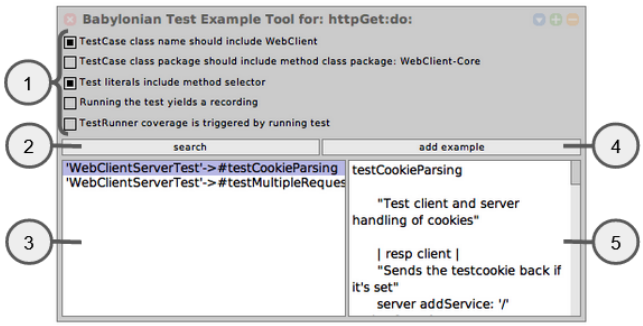

Figure 2: Example Mining tool to find relevant tests and add them as an example. Includes a pane to change search parameters (1), a button to trigger the search (2), a pane that displays found tests (3), a button to add the currently selected test as an example (4), and a pane that displays the code of the currently selected test (5).

4.2 From Debugging to Examples

If an error occurs in the WebClient, a debugger is triggered. The debugger could then be used to add an example to the current method, in order to further explore the error in the code browser or to document the error. It would of course also be possible to intentionally debug a piece of code to add as an example.

One possible scenario for this could occur while using Scamper, a Squeak/Smalltalk web browser. The developer wants to open a web page in Scamper, but an error occurs in the #defaultUserAgent

method of the debugger, causing a debugger to appear, which can be seen in Figure 3. In this scenario, the error was intentionally included by the programmer and is easy to fix, but it is still possible to add the method invocation of the debugger as an example.

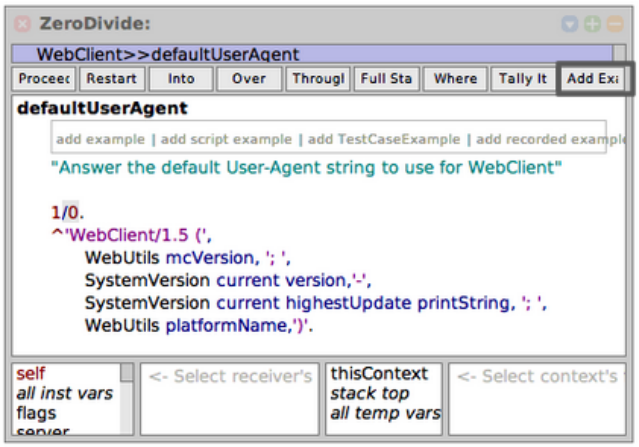

Figure 3: Example Mining debugger extension to add examples from the debugger. At the right side of the button pane a button was added that adds the current receiver and arguments as an example to the currently debugged method.

4.3 From Traces to Examples

If a programmer wants to add an example to the # authenticate :from: method of the WebClient, creating an example from scratch can be difficult. A correctly configured WebClient as well as WebRequest and WebResponse would be needed. However, the method comment tells us when the method is invoked: The method is called if an HTTP 401 'Unauthorized' error code is encountered. This means if we try access a web page without proper authentication, the target method will be called.

One easy way to access web pages is to open them in a browser. Scamper, a web browser implemented in Squeak/Smalltalk, allows us to do that. Instead of creating all objects needed for example from scratch, we could record all invocations of the target method while trying to access a web page without the needed authentication in Scamper.

First, a code browser needs to be opened for # authenticate :from:, our target method. Then an Example Mining recording tool for the method needs to be opened. Toggling the recording will turn it on, signaled by the circle element switching from grey to red. While the tool is recording, we open Scamper and put in the URL of a web page that we lack proper credentials for. Scamper will display the according 401 page, which means we can stop the recording. All recordings will now be displayed, which should be exactly one: the invocation caused by Scamper, which can be seen in Figure 4. This recording can be previewed and added as an example to the target method.

5 RELATED WORK

We discuss relevant concepts from testing and debugging.

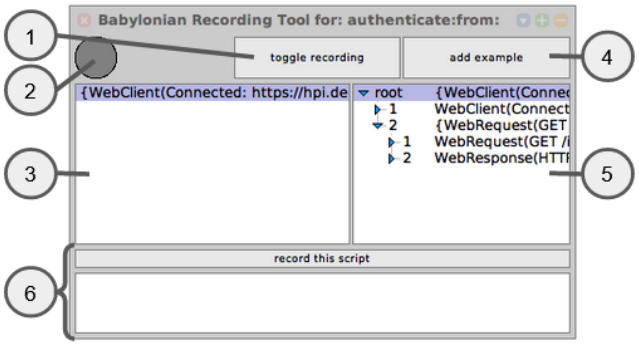

Figure 4: Example Mining tool to record traces and add them as examples. Includes a button toggle whether recording is active (1), a color-changing indicator whether recording is active (2), a pane that display all recordings (3), a button to add the currently selected recording as an example (4), and an explorer view of the objects of the selected recording (5). At the bottom of the tool a button and code field can be found that allow recording the script written in the code field without recording system behavior (6).

As argued above, tests can be a mining source for examples [4, 6]. To mine examples from tests, programmers first have to identify suitable tests. Some similar projects such as the Debugging into Examples tool record traces of all test executions to determine which tests execute which methods [30]. Similar precision can be achieved with our trace-based Example Mining tool. Our test-based mining tool uses heuristics instead. Similarly, general techniques from test selection and prioritization can help find tests that are relevant for certain sections of code [18, 22]. Some test prioritization techniques use the test coverage to prioritize tests and do so in a time-effective manner [20]. Such techniques could improve the filtering of tests in our test-base Example Mining tool. Another related, coverage-based technique may be used to find test cases relevant for a method. The technique can determine groups of tests cases relevant for a class, method, or theoretically even statement [10, 33].

For some methods, there might not be unit tests suitable to become examples but only integration tests that happen to execute the method. Related techniques can generate unit tests from system tests [5]. Our proposed tool might also benefit from this technique. Currently, our proposed tool would add a test-based example for the integration test. While this illustrates the method, it also might take longer to run and be more difficult to comprehend than an isolated unit test.

Debugging research is relevant for our tools, with regard to how to make information on concrete executions easy to access as well as with regard to techniques for handling traces. Visualization techniques in debuggers might be integrated into our tool to make it easier for programmers to select invocations to be converted to examples [2, 15, 19]. Similar to our tools, a lot of debugging tools either work on traces or generate them [8, 25]. As we use traces to repeat the original execution, capture-and-replay mechanisms in particular are relevant for our approach [1, 3, 5, 13, 24]. For instance,

our trace-based Example Mining tool currently only records the invocation of methods. To make the resulting example self-contained, any interactions with the environment should be mocked. That might be accomplished using record-and-replay techniques that record such interactions and create mock objects or methods to make replays repeatable [13, 24, 29].

6 CONCLUSION AND FUTURE WORK

To enable developers to reuse existing information to create, this paper introduced several ideas to mine examples from existing sources. Three sources in particular were focused on. The first source for examples described was tests, as tests both provide all information to run the test code and usually also fulfill a documenting purpose. The second source was debugging sessions that allow the developer to spontaneously create examples based on the information of the method invocation currently explored in a debugger. The third and last source focused on in this paper was traces of the system itself, allowing programmers to record all invocations of the method they are working on and add those recordings as examples.

Future Work. There are several aspects of the concepts and tools of this paper that can be expanded upon. Aside from evaluating the tools with example use cases and metrics, a user study could show if and how the tools are useful. The tools themselves could be expanded in several ways, while the underlying concepts could also be transferred to and implemented in other programming environments.

A user study that investigates the usage of the tools to see in which situations example mining is chosen over creating an example from scratch could yield further insights. This might also discover new aspects that the tools should support.

Several usability improvements could be made to the tools such as allowing further exploration of the recordings before adding them as an example. The heuristic search for relevant tests could still be improved. For example, techniques used in test prioritization could be applied to make it easier for programmer to select a fitting test as an example. Further, there are still other sources for examples that we have not yet covered. For instance, future tools might use example code from sources such as documentation, code annotations, or tutorials.

ACKNOWLEDGMENTS

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 449591262. We also gratefully acknowledge the financial support of HPI's Research School[1] and the Hasso Plattner Design Thinking Research Program[2].

REFERENCES

[1] Bowen Alpern, Ton Ngo, Jong-Deok Choi, and Manu Sridharan. 2000. DejaVu: deterministic Java replay debugger for Jalapeño Java virtual machine. In *Object Oriented Programming Systems Languages and Applications Conference, OOPSLA 2000, Minneapolis, MN, USA, October 15-19, 2000, Addendum to the proceedings*, James Haungs (Ed.). ACM, 165–166. https://doi.org/10.1145/367845.368073

[2] Ronald Baecker, Chris DiGiano, and Aaron Marcus. 1997. Software Visualization for Debugging. *Commun. ACM* 40, 4 (1997), 44–54. https://doi.org/10.1145/248448.248458

[3] Jong-Deok Choi and Harini Srinivasan. 1998. Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools - SPDT '98*. ACM Press. https://doi.org/10.1145/281035.281041

[4] Jonathan Edwards. 2004. Example centric programming. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 124. https://doi.org/10.1145/1028664.1028713

[5] Sebastian G. Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. 2006. Carving differential unit test cases from system test cases. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, Michal Young and Premkumar T. Devanbu (Eds.). ACM, 253–264. https://doi.org/10.1145/1181775.1181806

[6] Markus Gaelli. 2006. Modeling Examples to Test and Understand Software. (2006). https://doi.org/10.7892/BORIS.104524

[7] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley.

[8] Zhongxian Gu, Earl T. Barr, Drew Schleck, and Zhendong Su. 2012. Reusing debugging knowledge via trace-based bug search. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 927–942. https://doi.org/10.1145/2384616.2384684

[9] Christopher Michael Hancock. 2003. *Real-time Programming and the Big Ideas of Computational Literacy.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[10] Ferenc Horváth, Béla Vancsics, László Vidács, Árpád Beszédes, Dávid Tengeri, Tamás Gergely, and Tibor Gyimóthy. 2015. Test suite evaluation using code coverage based metrics. In *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST'15), Tampere, Finland, October 9-10, 2015 (CEUR Workshop Proceedings, Vol. 1525)*, Jyrki Nummenmaa, Outi Sievi-Korte, and Erkki Mäkinen (Eds.). CEUR-WS.org, 46–60. http://ceur-ws.org/Vol-1525/paper-04.pdf

[11] Tomoki Imai, Hidehiko Masuhara, and Tomoyuki Aotani. 2015. Shiranui: a live programming with support for unit testing. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 36–37. https://doi.org/10.1145/2814189.2817268

[12] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan C. Kay. 1997. Back to the Future: The Story of Squeak - A Usable Smalltalk Written in Itself. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '97), Atlanta, Georgia, October 5-9, 1997*, Mary E. S. Loomis, Toby Bloom, and A. Michael Berman (Eds.). ACM, 318–326. https://doi.org/10.1145/263698.263754

[13] Shrinivas Joshi and Alessandro Orso. 2007. SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions. In *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*. IEEE Computer Society, 234–243. https://doi.org/10.1109/ICSM.2007.4362636

[14] Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan O. Borchers. 2014. How live coding affects developers' coding behavior. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2014, Melbourne, VIC, Australia, July 28 - August 1, 2014*, Scott D. Fleming, Andrew Fish, and Christopher Scaffidi (Eds.). IEEE Computer Society, 5–8. https://doi.org/10.1109/VLHCC.2014.6883013

[15] Chris Laffra and Ashok Malhotra. 1994. HotWire - A Visual Debugger for C++. In *Proceedings of the C++ Conference. Cambridge, MA, USA, April 1994*, Doug Lea (Ed.). USENIX Association, 109–122. http://www.usenix.org/publications/library/proceedings/c++94/laffra.html

[16] Henry Lieberman and Christopher Fry. 1995. Bridging the Gulf Between Code and Behavior in Programming. In *Human Factors in Computing Systems, CHI '95 Conference Proceedings, Denver, Colorado, USA, May 7-11, 1995*, Irvin R. Katz, Robert L. Mack, Linn Marks, Mary Beth Rosson, and Jakob Nielsen (Eds.). ACM/Addison-Wesley, 480–486. https://doi.org/10.1145/223904.223969

[17] 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 ACM SIGPLAN International Workshop on Programming Experience, PX/17.2, Vancouver, BC, Canada, October 23-27, 2017*, Luke Church, Richard P. Gabriel, Robert Hirschfeld, and Hidehiko Masuhara (Eds.). ACM, 28–35. https://dl.acm.org/citation.cfm?id=3167109

[18] Yiling Lou, Junjie Chen, Lingming Zhang, and Dan Hao. 2019. Chapter One - A Survey on Regression Test-Case Prioritization. *Adv. Comput.* 113 (2019), 1–46. https://doi.org/10.1016/bs.adcom.2018.10.001

[19] Allen D. Malony, David H. Hammerslag, and David Jablonowski. 1991. Traceview: A Trace Visualization Tool. *IEEE Softw.* 8, 5 (1991), 19–28. https://doi.org/10.1109/52.84213

[1] https://hpi.de/en/research/research-school.html
[2] https://hpi.de/en/dtrp/

[20] Wes Masri and Marwa El-Ghali. 2009. Test case filtering and prioritization based on coverage of combinations of program elements. In *Proceedings of the International Workshop on Dynamic Analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009), WODA 2009, Chicago, IL, USA, July, 2009*, Ben Liblit and Andy Podgurski (Eds.). 29–34. https://doi.org/10.1145/2134243.2134250

[21] Sean McDirmid. 2013. Usable Live Programming. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld (Eds.). ACM, 53–62. https://doi.org/10.1145/2509578.2509585

[22] Dominik Meier, Toni Mattis, and Robert Hirschfeld. 2021. Toward Exploratory Understanding of Software using Test Suites. In *7th Programming Experience Workshop (PX/21)*. ACM. https://doi.org/10.1145/3464432.3464438

[23] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. 2020. 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, Virtual, November, 2020*. ACM, 1–17. https://doi.org/10.1145/3426428.3426919

[24] Alessandro Orso and Bryan Kennedy. 2005. Selective capture and replay of program executions. *ACM SIGSOFT Softw. Eng. Notes* 30, 4 (2005), 1–7. https://doi.org/10.1145/1082983.1083251

[25] Peter Phillips. 2010. Enhanced debugging with traces. *Commun. ACM* 53, 5 (2010), 50–53. https://doi.org/10.1145/1735223.1735240

[26] 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. *Art Sci. Eng. Program.* 3, 3 (2019), 9. https://doi.org/10.22152/programming-journal.org/2019/3/9

[27] Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, Fabio Niephaus, and Robert Hirschfeld. 2019. Implementing Babylonian/S by Putting Examples Into Contexts. In *Proceedings of the Workshop on Context-oriented Programming - COP '19*. ACM Press. https://doi.org/10.1145/3340671.3343358

[28] 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. *Programming Journal* 3, 1 (2019), 1. https://doi.org/10.22152/programming-journal.org/2019/3/1

[29] David Saff and Michael D. Ernst. 2004. Mock object creation for test factoring. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'04, Washington, DC, USA, June 7-8, 2004*, Cormac Flanagan and Andreas Zeller (Eds.). ACM, 49–51. https://doi.org/10.1145/996821.996838

[30] Bastian Steinert, Michael Perscheid, Martin Beck, Jens Lincke, and Robert Hirschfeld. 2009. Debugging into Examples. In *Testing of Software and Communication Systems, 21st IFIP WG 6.1 International Conference, TESTCOM 2009 and 9th International Workshop, FATES 2009, Eindhoven, The Netherlands, November 2-4, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5826)*, Manuel Núñez, Paul Baker, and Mercedes G. Merayo (Eds.). Springer, 235–240. https://doi.org/10.1007/978-3-642-05031-2_18

[31] Steven L. Tanimoto. 1990. VIVA: A visual language for image processing. *J. Vis. Lang. Comput.* 1, 2 (1990), 127–139. https://doi.org/10.1016/S1045-926X(05)80012-6

[32] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming, LIVE 2013, San Francisco, California, USA, May 19, 2013*, Brian Burg, Adrian Kuhn, and Chris Parnin (Eds.). IEEE Computer Society, 31–34. https://doi.org/10.1109/LIVE.2013.6617346

[33] Dávid Tengeri, Árpád Beszédes, Tamás Gergely, László Vidács, David Havas, and Tibor Gyimóthy. 2015. Beyond code coverage - An approach for test suite assessment and improvement. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 1–7. https://doi.org/10.1109/ICSTW.2015.7107476

[34] David M. Ungar, Henry Lieberman, and Christopher Fry. 1997. Debugging and the Experience of Immediacy. *Commun. ACM* 40, 4 (1997), 38–43. https://doi.org/10.1145/248448.248457

[35] Radhika D. Venkatasubramanyam and Sowmya G. R. 2014. Why is dynamic analysis not used as extensively as static analysis: an industrial study. In *1st International Workshop on Software Engineering Research and Industrial Practices, SER&IPs 2014, Hyderabad, India, June 1, 2014*, Rakesh Shukla, Anjaneyulu Pasala, and Srinivas Padmanabhuni (Eds.). ACM, 24–33. https://doi.org/10.1145/2593850.2593855