# The VIVIDE Programming Environment

## Connecting Run-time Information With Programmers' System Knowledge

Marcel Taeumel  Bastian Steinert  Robert Hirschfeld

Hasso Plattner Institute
University of Potsdam, Germany
{first.last}@hpi.uni-potsdam.de

## Abstract

Programmers benefit from concrete program run-time information during code-centric comprehension activities. Unfortunately, state-of-the-art programming environments distract programmers from their task-oriented thinking by forcing them to cope with (1) tool-driven run-time information access and with (2) tool-driven changing information views. However, current research projects address these problems with new concepts for capturing run-time behavior as needed and for organizing all information on-screen according to the programmers' mental model.

Unfortunately, there has been no attempt that tries to combine available solutions into one single approach. We propose a new concept for programming environments, which allow programmers to work in a task-oriented way: Run-time information is collected automatically using tests; Information is displayed consistently in self-contained editors arranged on a horizontal boundless tape. We illustrate practicability with an implementation in Squeak/Smalltalk.

We believe that such environments will allow programmers to explore program-related information without noticeable tool switches and hence context switches. Having this, the cognitive effort will be reduced and thus programmers will make fewer false conclusions and eventually save time.

***Categories and Subject Descriptors*** D.2.6 [*Software Engineering*]: Programming Environments—*integrated environments*

***General Terms*** Human Factors

***Keywords*** Programming environments, program comprehension, source code, dynamic analysis, concurrent views, navigation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*Onward! 2012,* October 19–26, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1562-3/12/10...$10.00

## 1. Introduction

De facto, programmers have to maintain large software systems and hence to understand source code of existing system parts [5][35]. When reading source code, programmers draw from their current system knowledge and reason about the underlying intent. Such code-centric program comprehension activities can be difficult because of non-descriptive identifiers. For example, names for classes, methods, and variables are part of a problem domain the programmer is often not fully knowledgeable with. Unfortunately, descriptive information is often unavailable. Hence, programmers simulate code execution mentally to understand the system part in detail.

Programmers benefit from reading information about program run-time behavior [14] to verify and extend their current system knowledge and hence reduce cognitive effort. By doing so, they can directly match abstract source code with concrete run-time information to understand the system as is. A common practice exposes program run-time state at selected points in execution time. Such log-based and breakpoint-based approaches rely on effective and efficient tool support.

Programmers benefit from using integrated programming environments [29][13], which integrate tools for browsing source code with tools for exploring program run-time. Unfortunately, working within these environments is rather distracting from the actual comprehension activity for two reasons: (1) programmers cannot access run-time information directly when questions arise and (2) information views are arranged in an unintuitive way requiring much effort for separating the known from the unknown. Fortunately, several research projects address these issues by introducing new concepts for capturing run-time behavior without programmers' attention [26][24][30][20] and new concepts that allow for arranging information in a way that better reflects programmers' mental model of the system [7][3][18]. To our best knowledge [6][31], there has been no attempt that tries to combine results from both fields into one single programming environment.

We think that programming environments should not only integrate tools for improved data exchange on a technical level, but furthermore should allow programmers to work in a task-oriented way and hence integrate their activities. If environments are more user-centric and less tool-driven, this will imply a better support in directly accessing, exploring, and understanding program run-time information to immediately answer questions that arise during comprehension activities.

In this paper, we make the following contributions:

- A concept for seamless integration of tools into one environment that provides views on both static and dynamic program-related information and hence that presents a system as one single unit.
- VIVIDE—our prototypical implementation of this concept that illustrates its practicability using an example comprehension task.

After this introduction to the problem domain, section 2 illustrates a motivating comprehension activity and recaps limitations of traditional programming environments in more detail while associating current research results that we build upon. As the main part, section 3 explains our conceptual solution including our notion of run-time data, an appropriate visualization metaphor, and a simple interaction model. To show practicability, section 4 presents VIVIDE—our research prototype of future programming environments—and explains several implementation details. Following that, section 5 summarizes related work and hence the way other research projects try to address those problems. Finally, section 6 draws conclusions and sketches open hypotheses.

## 2. A Motivating Scenario

Programmers are likely to get distracted from their current program comprehension activity when trying to access, explore, and understand program run-time behavior within modern programming environments like Eclipse[1] or Visual Studio[2]. They incidentally perform *context switches* in thinking because they cannot directly access and process the required information but first need to find a way for that within the environment. As a result, programmers tend to favor drawing from the own experience and to simulate the program control flow in mind. In the end, this increases the cognitive load for programmers and hence is prone to errors.

Within traditional programming environments, run-time information is accessed via *edit-compile-run* cycles: Programmers have (1) to decide where to write logging statements or to set breakpoints, (2) to recompile the program if necessary, and (3) to choose appropriate program entry points that seem to reach the desired point in execution. In each step, information is presented in different views that are loosely-coupled and hence require visual reorientation. Generally speaking, programmers get distracted constantly by the environment.

[1] http://www.eclipse.org

[2] http://www.microsoft.com/visualstudio/en-us

```
1  encodeOn: aDocument
2    "Encode the receivers attribute onto aDocument.
     Note that this implementation requires those two
     checks for true and false exactly the way they are
     here, to reliably encode boolean attributes in an
     XHTML−compliant way."
3
4    self keysAndValuesDo: [:key :value |
5      value == false ifFalse: [
6        aDocument
7          nextPut: Character space;
8          nextPutAll: key;
9          nextPutAll: '="'.
10       value == true
11         ifTrue: [aDocument nextPutAll: key]
12         ifFalse: [aDocument print: value].
13       aDocument nextPutAll: '"']].
```

**Listing 1.** A source code example taken from the Seaside 3.0 web framework. The method is part of the class `WAHtmlAttributes`, which is a hash map.

In this section, we illustrate the problem in detail by using a source code example from the web framework Seaside 3.0[3] [19][9]. The main reason for context switches seems to be twofold: (1) programmers need to make decisions about how to access run-time information and (2) programmers need to cope with changing views (or dedicated sub-tools like debuggers) within the environment. After that, we conclude what *integration* really should employ in programming environments.

### 2.1 A Non-descriptive Method

The programmer wants to understand the rendering system of Seaside and hence the way how HTML code is generated. After browsing several categories of classes that are promisingly named after domain concepts (i.e., "Seaside-Core-Rendering" and "Seaside-Core-Document"), she discovers the class `WAHtmlAttributes`. Basically, she is knowledgeable with this kind of hash map and how it is generally used in the domain of a web framework. But then, she reads through the method named `encodeOn:` as shown in listing 1. After trying to understand the comment (line 2), she wonders, in which situations the inner false-branch (line 12) is reached. Several questions arise:

1. How does `aDocument` look like at run-time?
2. In which scope is this method called?
3. Is the conditional branch in line 12 ever reached and which concrete implementation of `print:` is called then?

[3] http://www.seaside.st

***First attempt*** She wants to approach the first question with log-based debugging. Therefore, she adds a simple logging statement in line 3 because she knows that every object has a textual representation: "`Transcript show: aDocument.`"

Now she needs to decide how to run the framework in a way that `encodeOn:` is called. Tests seem to be appropriate because frameworks do not have any feasible main entry points per se like normal programs do. There is a test case called `WAHtmlAttributesTest` that seems suitable and that contains 8 tests. She chooses one named `testAt` but is unlucky with that choice because her method is not called. Eventually, the test `testAtPut` produces console output: "`a WAHtmlDocument`". This result is not satisfying and she needs more detailed run-time information to answer the first question.

*There are 2 out of 8 tests in the test case that do not cover this method. So there is a probability of 25 percent to choose the wrong test here. Actually, there are even 68 tests in the system that cover the method and hence are possible entry points.*

***Second attempt*** Now, she wants to try out breakpoint-based debugging to answer both the first and the second question. She sets a breakpoint at line 4. This time, she already knows which test to choose; the debugger stops the execution at the desired place. There, an *object explorer* appears and reveals detailed information about the state of `aDocument`. She gets distracted because the environment rearranges the position of the source code on the screen (Figure 1). Additionally, a view with the current *call stack* provides information about the calling methods and hence the scope that she was looking for.

After that, she continues the debugging session call-by-call until the inner false-branch (line 12) is reached. Fortunately, the current test does cover this branch. She discovers that this will always be the case if `value` is a text such as "foo". Then, the `print:` method of `WAXmlDocument` is called. Eventually, the whole algorithm in this method becomes clear.

***Reflection on Attempts*** Depending on the kind of run-time information that is needed, the log-based attempt could still be more efficient. Modern programming environments display information in many changing views (e.g., Eclipse's perspectives) and programmers may tend to favor focusing one single static console window over interacting with many appearing and disappearing support windows, e.g., caused by the debugger. That is why these two attempts are common practice for program comprehension activities. In the end, this trade-off can distract the programmer and hence be time-consuming.

### 2.2 Distraction 1: Meta-level Decisions

Whenever programmers want to access run-time information, they need to make two decisions that are *meta* to the comprehension question they are trying to answer:

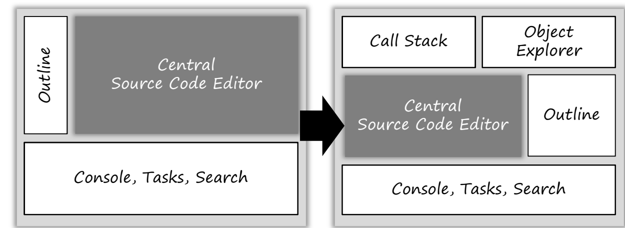



**Figure 1.** Perspectives in Eclipse can be noticeably different by default. Both screen dimensions are freely-used for layouting. Hence, information recalling can be hampered, e.g., when switching from the default perspective (left) to the debugging perspective (right).

1. Where and how to write (extensive) logging statements or to set (conditional) breakpoints?
2. Which (kind of) program entry point and hence input values to choose?

The example in section 2.1 illustrated, how the first decision can be approached iteratively depending on the kind of run-time information programmers are interested in. Experience can shorten the amount of time that is needed to come to this decision.

The second decision addresses *reachability* and *reproducibility*. Program comprehension is an iterative process and the compile-edit-run cycle, as mentioned above, is used for step-wise refinement and thus answering questions. Hence, programmers need to access reproducible results to understand a system part. The possibilities range from fully-manual (hence potentially not reproducible) application runs over replaying recorded user actions to the execution of scripted tests that cover the source code of interest. This decision needs to be tool-supported; otherwise a trial-and-error approach will be time-consuming like it is in traditional programming environments.

Current research results reveal that run-time information can be captured and provided automatically. Röthlisberger et al. [26] created SENSEO—an Eclipse plugin that embeds valuable information about program behavior efficiently collected during, e.g., test runs. Steinert et al. [30] extended the Squeak/Smalltalk programming environment to be able to open a debugger for any method of interest by automatically selecting and executing a covering test. Perscheid et al. [20] developed a time- and memory-efficient approach to access run-time information without programmers' attention ranging from lightweight call trees to detailed object states. Pothier et al. [22] also describe a practical approach to enhance traditional debugging with accessing additional run-time information in a scalable way.

### 2.3 Distraction 2: Changing Information Views

While browsing code, programmers get used to many views for information ranging from specific method sources over class definitions to general system outlines. Once run-time information needs to be accessed, this accustomed situation often changes visually [17], e.g.:

- The environment may rearrange all open views in the debugging mode to integrate new views like object explorers and call stacks. This "feature" is called *perspectives* in Eclipse (Figure 1).
- The console window may need to be resized manually in order to see all important outputs. Hence, source code views need to back off temporarily.
- The (graphical) program may appear in front of the environment and hide important parts of the source code.

In any case, programmers need to refocus and look for known and unknown information to be connected in mind. This process often involves many user interactions and hence can be time-consuming. A connection between run-time information, source code and hence programmer's system knowledge needs to be (re-)established. Modern programming environments do not support this efficiently.

Current research results reveal promising approaches for arranging program-related information on the screen. Bragdon et al. [3] created CODE BUBBLES—a new front-end for Eclipse that arranges simple, self-contained *bubbles* representing source code artifacts on a scrollable, two-dimensional canvas. There is a similar implementation for Visual Studio called DEBUGGER CANVAS[4]. Olivero et al. [18] created GAUCHO—a new front-end for Pharo/Smalltalk that arranges source code artifacts in nestable containers called *pampas'*.

### 2.4 The Problem: A Lack of Activity Integration

Until now, integrated programming environments have been represented a single application that hosts many supportive tools for reading, modifying, and debugging source code. Having this, fundamental programming tasks are supported consistently and hence ease the exchange of essential data between sub-tools like code editors, debuggers, command lines, and output consoles. An appealing graphical user interface underlines this picture of an apparently effective and efficient environment—But what should integration really employ?

> *"We shape our tools. And then our tools shape us."*
> —Marshall McLuhan

Traditional programming environments have one fundamental flaw: they are still just fancy text editors that provide ways to enrich only one central view on source code. Programmers have to learn how to map their current activity onto this inappropriate user interface metaphor [25]. When doing so, they are not working *within* the environment but use the environment to accomplish their task in a distracting way.

When comprehending programs, programmers benefit from accessing and processing both source code and run-time information. Hence, programming environments need to support data-centric activities and not just provide tool-centric opportunities. This means, that programmers should be able to freely-navigate within the space of all program-related static and dynamic information in a non-distracting way as needed.

## 3. A User-centric Programming Environment

In this section, we present our concept for programming environments that reduce the number of context switches when accessing run-time information during program comprehension activities. Within this environment, programmers stay focused in their task-/problem-oriented thinking and do not have to make tool-driven decisions that are distracting and hence time-consuming. A clear, consistent user interface abstracts from technical details and integrates with programmers' activities in a user-centric way by directly supporting answering questions in understanding whenever they arise.

Unger et al. identified three different types of *immediacy* [34] that programming environments should support to keep programmers focused on their task: temporal, spatial, and semantic immediacy:

- Temporal immediacy addresses the delay between performing an action and receiving a feedback in the environment.
- Spatial immediacy addresses the visual distance of related information on-screen.
- Semantic immediacy addresses the number of user interactions (e.g., mouse clicks) needed to access a desired information.

At first, we address the problem of *meta-level decisions* by explaining our notion of run-time information and how to capture and provide it automatically. This corresponds to temporal immediacy. Then, we address the problem of *changing views* and present a visualization based on a scrollable tape with embedded editors to display needed information in a simple, clear, and predictable way. Hence, spatial and semantic immediacy are ensured.

### 3.1 Capturing Example Run-time Information

Our notion of run-time information encompasses exemplified program behavior to support code-centric comprehension tasks. In the strict sense, we want to collect information about method calls—namely object states (i.e., callers, callees, arguments, results) and behavioral traces (i.e., call trees). We do not target concrete debugging scenarios where defects have to be found; the awareness of specific failing

[4] http://msdn.microsoft.com/en-us/devlabs/debuggercanvas

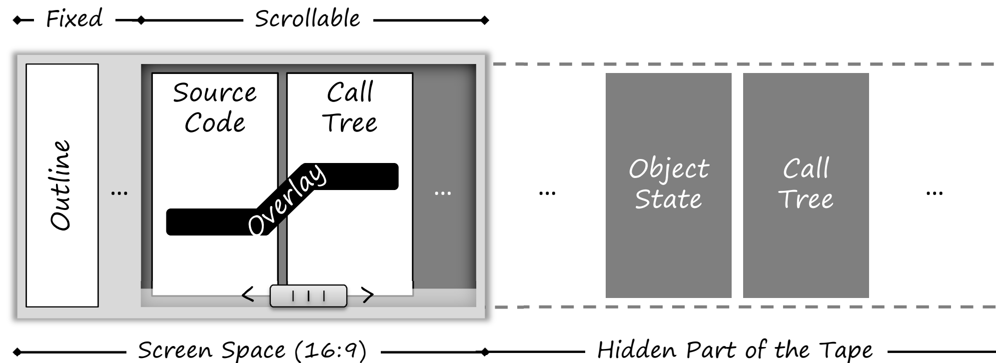



**Figure 2.** Our concept for integrated programming environments. Editors are arranged from the left to the right either in a fixed portion of the screen space or on a horizontal boundless tape that is accessible through a scrollable container. Overlays visualize relations between editors. Any kind of editor can be placed multiple times on the tape.

tests would be important for that [21]. Furthermore, examples of indifferent origin should help to map abstract source code to concrete program behavior and hence to verify and extend system knowledge at an exemplary but valuable level.

Tests are well-suited program entry points that produce representative control flows and hence valuable information about program behavior [30][20][21]. In fact, writing tests is known to be supportive during software development [2][1]. By having these defined program entry points, dynamic analysis techniques [6][24][20] are able to capture run-time information without requiring programmers' attention. Programmers can focus their comprehension activity and query run-time information directly when needed—assuming that tests cover the method of interest.

Test coverage is an important aspect in this approach. Normally, coverage is measured with the number of methods covered in percent, but considering conditional branches at sub-method level [8], a coverage of 100% could still not be sufficient. Generally speaking, the more tests there are that cover a method, the more run-time information can be provided and hence the more programmers' understanding can be supported. From a pragmatic perspective, any available information is valuable. Although the amount of accessible run-time information is not quantifiable in percent, an application of the well-known Pareto principle would imply that even 20% of accessible program behavior could help to understand the underlying intent to a degree of 80%. All in all, test should be used to capture example run-time information in programming environments.

There is a shift of responsibility considering run-time information access from programmers to the environment. Hence, tests need to be deterministic. Until now, reproducibility of run-time information has been more important from a programmer's perspective than from a technical perspective. Programmers may need to recall the same information several times during step-wise refinement of comprehension questions depending on their mental capacity. In our concept, programmers do not have to think of ways to achieve reproducible results anymore but implementations of such environments do have to. Direct information access means that programmers are aware of the number of, for example, mouse clicks they have to perform and hence they will notice how long the *response times* are, until the desired information becomes visible on the screen. The size of this time frame has an impact on when to lose focus on the current activity[5]. The problem is that dynamic analysis can be expensive considering time- and memory-consumption [15]. Hence, implementations of our concept need to pay attention to performance issues and may consider *partial tracing* approaches [32][23][20]. Therefore, reproducible results rely on deterministic tests.

All kinds of program comprehension questions that we address can be reduced to automatic capturing, querying, and post-processing of object states and behavioral traces in the context of a specific method call. Post-processing varies from simply accessing example data to aggregating all information for providing ranges of possible variations in a broader scope.

### 3.2 Displaying Source Code and Run-time Information

The visualization part in our concept tries to mask the presence of dedicated sub-tools and hence tries to combine source code and run-time information in a way that directly integrates with programmers' comprehension activities. For this, the desktop metaphor, which tries to imitate real-world artifacts and activities in graphical user interfaces, is considered as inappropriate because programming environments have no representations of artifacts or activities in the real-world [25].

We put each self-contained portion of information (e.g., a class' methods, an object state, or a call tree) into one rectangular view—called *editor*. Editors are arranged on a horizontal unbounded *tape* side by side. Connections between visible information are displayed via *overlays*.

[5] Shneiderman et al. [28, p. 445] argue that a response delay of 1 second does still not distract the user from simple and frequent tasks.

***Horizontal Tape*** Modern wide-screen monitors offer an image ratio of 16:10 or 16:9. Having this, the primary (since largest) screen axis is the horizontal one and programmers need to think about how to make efficient use of the available screen space. Since source code lines are rarely longer than 100 characters, this kind of information tends to spread along the vertical axis leaving much whitespace to the left or to the right. Traditional programming environments surround this central code area with freely-arrangeable views for, e.g., system outlines, documentation, or run-time information to make use of whitespace. This leaves both screen dimensions open for different kinds of information that programmers may have to look for.

Our concept proposes a *horizontal unbounded tape* that is embedded into a scrollable area as shown in figure 2 to make efficient use of wide-screen monitors. On this tape, editors are freely-arrangeable from the left to the right. This assigns a clear level of information granularity to each screen axis: the horizontal is reserved for different kinds of information (e.g., source code, call trees, object states) and the vertical exposes details for each kind (e.g., chronologically ordered call nodes). Hence, programmers should be able to recall information more quickly.

Besides the tape, part of the screen space is reserved for editors that should always be visible: the *fixed area*. Having this, the environment organizes information in a two-level hierarchy:

1. Is the information always visible or potentially hidden?
2. Is the information to the left or right of the current view?

Still, these constraints allow for an unrestricted exploration of the system while avoiding programmers to get distracted when positioning information on the screen. Additionally, new editors that are about to appear can be positioned in a more predictable manner for programmers.

***Simple Editors*** Each editor contains details for one primary kind of information. This can be displayed in a central list, table, tree, or other visualization. For example, class editors can show a list of open methods, system overviews can show tree-like outlines of captions, call trees can show concrete behavioral traces, object explorers can compare object states before and after an exemplary method call.

In addition, pop-up menus and tooltips can reveal other (secondary) kinds of information that are directly associated. For example, the tooltip for each node in a call tree can show the called method's source code. Having this, programmers can directly connect abstract source code with concrete run-time information and hence verify/extend their current system knowledge.

Different to views in traditional programming environments, all editors in our concept are of equal priority for program comprehension activities. Editors for source code are not more important than editors for run-time information and vice versa. There is no central source code editor per se.

***Connecting Overlays*** All editors on the tape are arranged side by side and hence do not indicate any relation between them visually. However, there are such relations, for example:

- the navigation history caused by step-wise refinement during the comprehension process
- direct connections between visible source code and corresponding run-time information

Besides the tape and editors, our concept uses *overlays* as a third technique to illustrate those relations and hence allows programmers to recall information more quickly while reducing the cognitive load.

### 3.3 Interacting Within the Environment

In our concept, direct access to run-time information means to provide supportive information to an arising comprehension question with as few user interactions (e.g., mouse clicks) as possible. Having this, we believe that programmers will keep on exploring the program itself instead of exploring the environment. To achieve this, our concept considers common *starting points* for comprehension activities and *simple queries* using a consistent vocabulary in pop-up menus to navigate between different kinds of information.

***Starting Points*** Code-centric program comprehension starts with source code reading and looking for promising beacons [35]. This could mean to browse overviews of system parts or detailed sources of methods. To achieve this, editors that show the appropriate information need to be directly accessible using search mechanisms. In the first place, a text-based search is sufficient because programmers start with looking for identifiers (e.g., class names or method signatures) that seem to correspond with domain concepts when exploring systems [29]. When getting more knowledgeable in a system, this search could be extended to make use of run-time information.

However, this concept tries to reduce the complexity of queries. Programmers should not have to translate rather complex comprehension questions into the complicated vocabulary of environments.

***Simple Queries*** Simplicity is important when directly accessing run-time information. At first, programmers need to transform their question into one out of three elementary purposes:

- Browse Code ... to navigate to static information
- Explore Object ... to navigate to dynamic information
- View Trace ... to navigate to dynamic information

This transformation is supposed to be straightforward because programmers are aware of these fundamental kinds of information in object-oriented programs. This simple vocabulary should be visualized with *pop-up menus* and integrated

into all editors consistently. By doing so, programmers are free to decide whether, for example, given pieces of run-time information benefit from additional run-time data or source code.

Methods that are covered by many tests provide many examples for run-time behavior. Hence, all accessible information should be aggregated to make it perceivable for programmers. More sophisticated queries would be necessary. For example, such *ranges of variations* in object states or behavioral traces could be accessible through pop-up menus that allow for interaction that is more extensive. In general, any visualization should follow the *Visual Information Seeking Mantra* as proposed by Shneiderman [27]: "Overview first, zoom and filter, then details-on-demand." Eventually, any comprehension activity can include concrete examples even when starting with an overview of all available run-time information about the methods of interest.

### 3.4 Concept Summary

Our concept has no explicit notion of a compile-edit-run cycle because tests are executed and traced without programmers' attention. All information is presented using self-contained editors on a horizontal unbounded tape. Overlays visualize relations between visible information. Clearly defined starting points in conjunction with simple queries for both source code and run-time information avoid time-consuming context switches. Hence, temporal, spatial, and semantic immediacy [34] are ensured. Programmers are supported to focus program comprehension activities instead of making distracting tool-driven meta-level decisions.

## 4. The VIVIDE Programming Environment

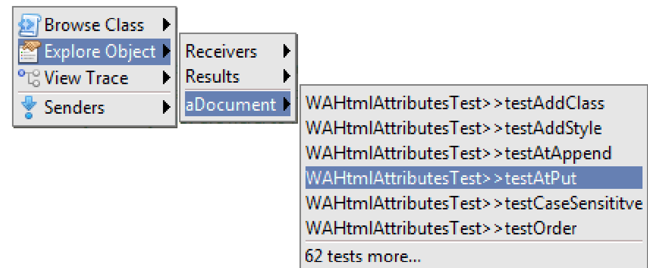
This section presents VIVIDE—our research prototype that implements our concept for task-oriented programming environments, which directly integrate with programmers' comprehension activities. It is written in Squeak/Smalltalk[6] and includes many of the ideas presented in section 3: the horizontal tape for efficient screen space usage, several editors for browsing source code and run-time information, and pop-up menus for simple navigation within the space of available information. At the moment, there are no overlays that make relations between editors explicit.

At first, we replay the motivating scenario from section 2.1—but this time using our prototype. The resulting screen contents are shown in figure 3. After that, we explain several implementation details about how to keep the response times low when using our prototype.

### 4.1 A Non-descriptive Method—Revised Attempt

The programmer wants to understand the rendering system of Seaside and hence the way how HTML code is generated. Therefore, she browses several identifiers of class categories, classes, and methods with the *package viewer* (A). Then, she discovers the class `WAHtmlAttributes` in "Seaside-Core-Document". Basically, she is knowledgeable with this kind of hash map and how it is generally used in the domain of a web framework. But then, she reads through the method named `encodeOn:` shown in a tooltip and opens a *source code editor* (B). After trying to understand the comment, she wonders, in which situations the inner false-branch is reached.

In a first step to comprehend this method, she explores an instance of `aDocument` by choosing the test `testAtPut` that covers this method via an attached pop-up menu:




As expected, an *object explorer* (C) appears to the right of the source code on the tape and displays example run-time information about how `aDocument` is modified during such a method call. There, she discovers that the text "foo" is written into the object.

Then still without pondering about debugging and program entry points, she opens an example *call tree* (D) using the pop-up menu again (i.e., "View Trace"). There, she investigates the calling context and discovers, that the inner false-branch is called, too.

After recalling all visible information about the method on the tape, the algorithm becomes clear. As a last step, she makes use of *embedded links* and sees that the `print:` method of the class `WAXmlDocument` is called in the inner false-branch:

```
value == true
  ifTrue: [aDocument nextPutAll: key]
  ifFalse: [aDocument print: value].
                      WAXmlDocument>>print:
```

At the and, she is still focused on the task to understand the rendering system of Seaside because all required program-related information were directly accessible.

***Reflection on Attempt*** The comprehension activity is supported with well-defined starting points. Beacons in the source code are enhanced with simple queries to directly navigate to helpful run-time information. This information is captured and presented immediately, hence keeping the programmer focused. During the whole activity, all editors on the horizontal tape allow for externalizing the mental model of the program.

[6] `http://www.squeak.org`

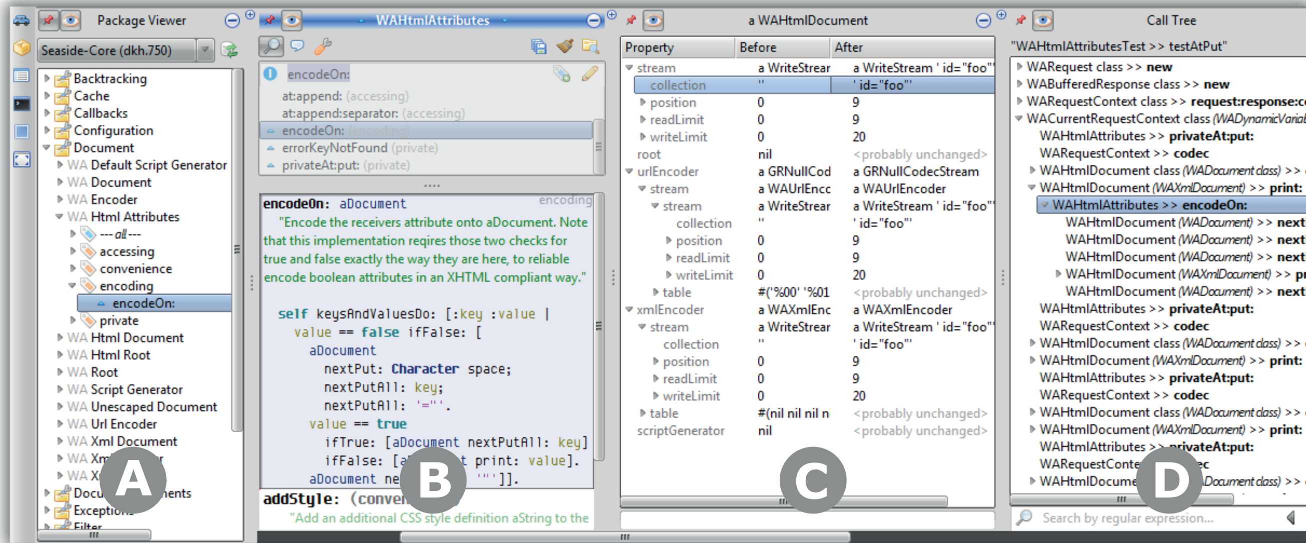



**Figure 3.** VIVIDE—our research prototype that implements our concept for new programming environments. The example session shows: (A) a system outline in the fixed area, (B) source code of the class `WAHtmlAttributes`, (C) an object state comparison for `aDocument` before and after the call of `encodeOn:`, (D) a call tree showing an exemplary calling context.

### 4.2 Keeping Response Times Low

As proposed in the concept section 3.1, VIVIDE aims to provide temporal immediacy and captures run-time information using test runs as appropriate application entry points. The approach is based on the time- and memory-efficient implementation of Perscheid et al. [20]. It consists of two subsequent steps:

**Shallow Analysis** The first step executes all relevant tests and traces few information about every method call to be able to reconstruct the whole control flow in chronological order. The resulting lightweight call tree can directly be viewed in the corresponding editor.

**Refinement Analysis** The second step is repeated several times and hence relies on deterministic control flows. Tests are executed again and selected nodes in associated call trees are enriched with object state copies to be used in editors, e.g., the object explorer.

Their implementation is optimized for tracing one test at a time. Since VIVIDE provides run-time information for method calls, it has to trace all tests that cover a specific method. After adapting the algorithm for this use case, our prototype provides appropriate response times below 1 second on average, which is sufficient to avoid user frustration for such simple and frequent tasks [28, p. 445].

Seaside 3.0.5 includes 696 tests that cover only 34% of all 5104 methods. However, if a method is covered, 20 tests will pass this method on an average. This means, that about 20 tests will have to be traced in the shallow analysis step unless already cached. Measurements[7,8] revealed that this takes 300 ms on average for all tests that cover a method. Hence, programmers are not distracted by a noticeable unresponsive user interface.

The coverage information is collected based on the approach of Steinert et al. [30] in a separate run of all available tests creating appropriate references between tests and methods. Both approaches use METHOD WRAPPERS [4] to modify program behavior accordingly. Any method call can be intercepted and tracing code can be inserted.

The editors and the tape in VIVIDE are implemented with the Morphic framework [16]. Conceptually, every graphical object (e.g., windows or buttons) on the screen consists of hierarchically composed morphs. For performance reasons, this object-oriented approach is sometimes softened, e.g., for text rendering, where single glyphs are directly processed using low-level drawing operations. Accordingly, VIVIDE optimizes smooth scrolling operations on the unbounded tape with caching a static representation of all editors in a single picture. Having this, programmers can easily notice, when new editors appear, and fluently recall all visible information.

## 5. Related Work

This section summarizes several related projects that aim for improving programming activities with new or extended environments.

[7] Intel Core i5-750, 4 GB RAM, Windows 7 Professional SP1, Squeak 4.2, Squeak VM 4.1.1

[8] VIVIDE is a 32-bit single core application.

The SELF environment [33] directly supports the programmer during exploration and comprehension activities of program behavior. Like in Squeak/Smalltalk, program run-time is omnipresent in this environment. Every object provides an *outline* that reveals information about its run-time state and known methods, i.e., the source code. However, this fine-grained access lacks overview capabilities. VIVIDE provides reasonable starting points for comprehension tasks to give programmers an overview of the system part. In subsequent steps, they can explore more detailed information up to single methods and objects.

Bragdon et al. created the CODE BUBBLES [3] environment, which arranges portions of program-related information using *bubbles* on an unlimited, two-dimensional, horizontal canvas. Programmers can arrange bubbles manually as needed during comprehension activities. VIVIDE provides a more guided approach that allows programmers to arrange *editors* with fewer interactions and to more easily perceive information because of the well-defined roles of the horizontal and vertical screen axes.

Olivero et al. created GAUCHO [18], which arranges source code artifacts using fine-grained *shapes* representing packages, classes, and methods in nestable containers. Hence, an unlimited space to store information is available. VIVIDE provides a two-level hierarchy to organize information using the *fixed area* and the left-right-arrangement of editors. We believe this is sufficient to group information while requiring less user interactions for navigation. We think that a hierarchy with more levels is not beneficial and can impede information recalling and hence increase the cognitive effort.

Karrer et al. created a new graphical interface for Xcode[9] called STACKSPLORER [10]. While editing source code files, the environment provides fan-in and fan-out information about methods. This allows for navigating possible control flows with ease. However, they only make use of static analysis techniques and do not consider concrete run-time information as VIVIDE does.

Ko et al. introduced WHYLINE [12], a tool for supporting debugging activities with the idea of directly answering questions that programmers have in mind. These questions simply address exploration of object states, e.g., why instance variable X was set to Y. VIVIDE provides simple queries to navigate source code and run-time information, hence also avoids the need for programmers to make complicated translations between their intent and available tool features.

Kersten et al. created MYLYN[10], a plugin for Eclipse that makes tasks being treated first-class in the environment. It is based on a degree-of-interest model [11] and allows programmers to group task-related artifacts to tidy up the user interface. This supports programmers to keep focus on relevant system parts. In a similar way, VIVIDE supports this with a clear arrangement of editors on the tape.

[9] `http://developer.apple.com/xcode`

[10] `http://www.eclipse.org/mylyn`

## 6. Conclusions

Run-time information supports programmers in understanding the intent of abstract source code. When working with traditional programming environments, programmers are likely to get distracted from their task-oriented thinking and incidentally perform context switches because tool-driven meta-level decisions impede direct information accessing and processing.

In this paper, we proposed a new approach for programming environments that directly integrate with programmers' comprehension activities based on available research results. In our VIVIDE prototype, we showed that programmers do not notice tool switches and hence avoid context switches in thinking. They can work in a task-oriented way because:

1. Having coverage information, tests are used to capture run-time behavior (e.g., call trees and object states) without programmers' attention on demand in a time- and memory-efficient way.
2. Static (e.g., source code and system models) and dynamic (e.g., behavioral traces) information is arranged intuitively in self-contained editors on a horizontal unbounded tape. Simple queries allow programmers to navigate freely within the space of program-related data.

We believe that if programmers work within such programming environments, they will come to fewer false conclusions and hence will make fewer mistakes during comprehension activities. Hence, we believe that they will accomplish tasks in a shorter period of time.

## References


[1] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2003.

[2] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2004.

[3] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola Jr. Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems*, pages 2503–2512. ACM, 2010.

[4] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the Rescue. *ECOOP 98—Object-Oriented Programming*, pages 396–417, 1998.

[5] T. A. Corbi. Program Understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[6] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A Systematic Survey of Program Comprehension Through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, September/October 2009.

[7] R. DeLine and K. Rowan. Code Canvas: Zooming Towards Better Development Environments. In *Proceedings of the 32nd International Conference on Software Engineering - Volume 2*, pages 207–210. ACM/IEEE, 2010.

[8] M. Denker, S. Ducasse, A. Lienhard, and P. Marschall. Sub-Method Reflection. *Journal of Object Technology*, 6(9):231–251, October 2007.

[9] S. Ducasse, L. Renggli, C. D. Shaffer, R. Zaccone, and M. Davies. *Dynamic Web Development with Seaside*. Lulu, 2010.

[10] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. Borchers. Stacksplorer: Call Graph Navigation Helps Increasing Code Maintenance Efficiency. In *Proceedings of the 24th Symposium on User Interface Software and Technology*, pages 217–224. ACM, 2011.

[11] M. Kersten and G. C. Murphy. Mylar: A Degree-of-Interest Model for IDEs. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, pages 159–168. ACM, 2005.

[12] A. J. Ko and B. A. Myers. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering*, pages 301–310. ACM/IEEE, 2008.

[13] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks. *IEEE Transactions on Software Engineering*, 32(12): 971–987, December 2006.

[14] T. D. LaToza and B. A. Myers. Developers Ask Reachability Questions. In *Proceedings of the 32nd International Conference on Software Engineering - Volume 1*, pages 185–194. ACM/IEEE, 2010.

[15] B. Lewis. Debugging Backwards in Time. In *Proceedings of the 5th International Workshop on Automated Debugging*, Ghent, Belgium, September 2003.

[16] J. H. Maloney and R. B. Smith. Directness and Liveness in the Morphic User Interface Construction Environment. In *Proceedings of the 8th Symposium on User interface and Software Technology*, pages 21–28. ACM, 1995.

[17] G. C. Murphy, M. Kersten, and L. Findlater. How are Java Software Developers Using the Elipse IDE? *IEEE Software*, 23(4):76–83, July/August 2006.

[18] F. Olivero, M. Lanza, M. D'Ambros, and R. Robbes. Enabling Program Comprehension through a Visual Object-focused Development Environment. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*, pages 127–134. IEEE, 2011.

[19] M. Perscheid, D. Tibbe, M. Beck, S. Berger, P. Osburg, J. Eastman, M. Haupt, and R. Hirschfeld. *An Introduction to Seaside*. Software Architecture Group, Hasso-Plattner-Institut, 2008.

[20] M. Perscheid, B. Steinert, R. Hirschfeld, F. Geller, and M. Haupt. Immediacy through Interactivity: Online Analysis of Run-time Behavior. In *Proceedings of the 17th Working Conference on Reverse Engineering*, pages 77–86. IEEE, 2010.

[21] M. Perscheid, M. Haupt, R. Hirschfeld, and H. Masuhara. Test-driven Fault Navigation for Debugging Reproducible Failures. *Journal of the Japan Society for Software Science and Technology*, 29, 2012.

[22] G. Pothier and E. Tanter. Back to the Future: Omniscient Debugging. *IEEE Software*, 26(6):78–85, November 2009.

[23] D. Röthlisberger, M. Denker, and E. Tanter. Unanticipated Partial Behavioral Reflection: Adapting Applications at Runtime. *Computer Languages, Systems & Structures*, 34(2-3): 46–65, July–October 2008.

[24] D. Röthlisberger, O. Greevy, and O. Nierstrasz. Exploiting Runtime Information in the IDE. In *Proceedings of the 16th International Conference on Program Comprehension*, pages 63–72. IEEE, 2008.

[25] D. Röthlisberger, O. Nierstrasz, and S. Ducasse. Autumn Leaves: Curing the Window Plague in IDEs. In *Proceedings of the 16th Working Conference on Reverse Engineering*, pages 237–246. IEEE, 2009.

[26] D. Röthlisberger, M. Hárry, W. Binder, P. Moret, D. Ansaloni, A. Villazón, and O. Nierstrasz. Exploiting Dynamic Information in IDEs Improves Speed and Correctness of Software Maintenance Tasks. *IEEE Transactions on Software Engineering*, 38(3):579–591, May 2012.

[27] B. Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proceedings of the Symposium on Visual Languages*, pages 336–343. IEEE, 1996.

[28] B. Shneiderman and C. Plaisant. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 5th edition, 2009.

[29] J. Sillito, G. C. Murphy, and K. De Volder. Asking and Answering Questions During a Programming Change Task. *IEEE Transactions on Software Engineering*, 34(4):434–451, July 2008.

[30] B. Steinert, M. Perscheid, M. Beck, J. Lincke, and R. Hirschfeld. Debugging into examples. *Testing of Software and Communication Systems*, pages 235–240, 2009.

[31] M. A. Storey. Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 181–191. IEEE, 2005.

[32] E. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. In *Proceedings of the 18th Conference on Object-oriented programing, systems, languages, and applications*, pages 27–46. ACM SIGPLAN, 2003.

[33] D. Ungar and R. B. Smith. Self. In *Proceedings of the 3rd Conference on History of Programming Languages*, pages 9–1–9–50. ACM SIGPLAN, 2007.

[34] D. Ungar, H. Lieberman, and C. Fry. Debugging and the Experience of Immediacy. *Communications of the ACM*, 40 (4):38–43, April 1997.

[35] A. Von Mayrhauser and A. M. Vans. Program Comprehension during Software Maintenance and Evolution. *Computer*, 28 (8):44–55, 1995.