Exploring Developer Preferences for Visualizing External Information Within Source Code Editors

Xinhong Liu
Department of Computer Science
University of British Columbia
Vancouver, Canada
xinhliu@cs.ubc.ca

Reid Holmes
Department of Computer Science
University of British Columbia
Vancouver, Canada
rtholmes@cs.ubc.ca

***Abstract*—**
Developers increasingly rely on external tools and services which causes development information to be scattered across different information silos. To access this information, developers need to access different applications, presentation files, and web services. This paper investigates mechanisms for incorporating external information sources into Integrated Development Environments using visual mechanisms to support common software development activities. Through a developer survey and a small experiment we find that developers prefer minimal representations for incorporating external information sources into their source code editors, and that they are able to use this information when performing their development tasks.

***Index Terms*—Integrated Development Environments, external information, actionable information, information overload**

I. Introduction

While building software, developers use many sources of external information (e.g., [1]–[5]). According to a survey conducted by JetBrains in 2020[1], in addition to Integrated Development Environments (IDEs) in which developers write and modify their code, 85% of developers use source code collaboration tools, and nearly 50% of developers use issue trackers, Continuous Integration (Continuous Integration (CI)), or Continuous Development (Continuous Delivery (CD)) tools. Other categories of standalone tools, such as static analysis tools and code review tools, also see significant industrial usage. Developers choose to use these tools and services because they each help them accomplish their work tasks, even if they are not directly integrated with one another or with their source code editor.

These disparate tools improve the development process, but impose other overheads on the developer as they work to integrate the information provided by those tools [6], [7]. This overhead arises because while these external tools act on the developer's source code, they usually execute independently, including on developers' computers or remote servers, and usually store their output separate from the source code itself [8], [9]. This means to analyze or use the output of these tools developers need to leave their code editor, navigate to the tool, and find the information within the tool they need. For example, to increase test coverage, a developer first needs to leave their IDE to open a web browser and navigate to the web-based interface for their coverage tool, which was generated by their continuous integration service. Second, they must navigate through the results to find uncovered lines. Third, they need to open the corresponding source file in their local IDE for the uncovered lines and finally develop new unit tests for these lines. While these tasks are not hard, they had to navigate two disconnected hierarchies (the coverage report on the server and the source code files in the IDE) and manually connect them (by the file names the developer was looking at in each hierarchy) before the developer could complete their task.

While many tools and services can be hosted locally, engineering teams increasingly rely on cloud-based services [10]. One commonality between most of these kinds of tools is the source code itself: code acts as an implicit anchor that can be used to tie the results from an external tool back to the source code the developer is working on for their task.

Although a number of tools have explored techniques for linking information back to source code (e.g., Bridge [11], Hipikat [12], , CodeBook [9]), and others have used inline source code representations for displaying results in code editors (e.g., HATARI [13], Whyline [2], PerformanceHat [5]), few papers have examined differences in the *representations* themselves which is the focus of this work.

In this paper, we explore developer preferences and performance for different visual representation mechanisms for integrating information from external information silos into source code editors. Our primary concern with this research was to learn, from real developers, how they perceive these visual representations, and whether they helped (or hindered) their development tasks. By focusing directly on the representations, this work aims to provide evidence tool designers can consider to improve how they integrate with source code editors. To understand this, we surveyed 21 developers to learn their preferred integration representations for a variety of different software development tasks.

Ultimately, we learned that developers do value having external information embedded within source code editors but prefer lightweight representations that enable more comprehensive on-demand investigation over more heavyweight representations. We created a functional technology probe

[1]The State of Developer Ecosystem in 2020
https://www.jetbrains.com/research/devecosystem-2020/

using a subset of these representations and evaluated these with a controlled user experiment with 8 participants to see how these representations impacted them as they performed four development tasks. Through this experiment we found that developers were able to quickly and effectively use external information through these integrated representations.

The contributions of this paper are:

1) A description of common visual representations for integrating information into source code editors.
2) A survey with 21 developers examining the relative strengths and weaknesses of these representations for common development tasks.
3) A controlled experiment with 8 participants demonstrating that developers can use inline representations to complete concrete development tasks with fewer actions and context switches.

II. Integrating External Information

The goal of this paper is to determine how to best provide actionable external information through software visualizations integrated into source code editors without overwhelming developers.

A. Integration design principles

To accommodate integrating the diversity of external information sources into source code editors, we detail several design principles that strive to balance the actionability of the integrated information with the risk of that information becoming overwhelming for the developer. This analysis is restricted to information needs that can be associated with specific project artefacts (e.g., lines, methods, or files) rather than more general information needs. For example, while this analysis includes code coverage (which is associated with lines) or test history (which is associated with test methods), it excludes project scheduling or marketing.

A natural concern when considering integrating a wealth of external information into source code editors is that the external information could distract from the source code itself. Avoiding overwhelming developers with information that is not relevant to their tasks is important because to developers *"code is king"* [14]. We used the following four presentation principles to guide our visual design decisions in order to try to maximize the informativeness of the external information while simultaneously minimizing its overwhelmingness. The principles were identified by manually examining a variety of existing source code editor plugins and identifying common positive properties of their visual representations.

These principles are not meant to be prescriptive, but to provide different dimensions for designers to consider when thinking about how to integrate external information into development environments. While these principles are described as actions a designer *should* take, these steps are only relevant if a designer aims to support a specific principle.

a) Non-intrusiveness.: It is important that external information not distract developers. The main focus within code editors is always the source code. Minimal representations should be preferred; if a larger view is needed, it should be hidden until it is contextually activated. Colours should be chosen to not overlap with those used in the code editor already (such as those used for syntax highlighting).

b) Intuitive expressiveness.: External information sources each have their own semantics and these should be considered in their representations. Wherever possible, graphic representations and colours should be chosen that fit with the information being displayed. For example, a green ✓ could represent code that is "covered by a passing test" while a red ✗ could represent "covered by failed tests".

c) Expandability.: External information sources encode rich sets of data. While representations should not be intrusive, these abstractions will naturally encode less information than the full information source. Any representation used should make it possible to expand the information should the developer want to know more, or make it possible to directly access the information in the external representation, if needed.

d) Uniqueness.: The presentation of information should be distinguishable from other information. While external information sources should encode their representation in intuitive ways, they should be cognizant of how other sources might also encode themselves such that they avoid confusion.

B. External information needs

Developers frequently refer to external information such as version control, build status, test results, and coverage. Here we categorize external information usage according to their purpose from existing literature. Each kind of external information is usually associated with a specific element type (e.g., a variable, statement, line(s), block, method/function, or class/file) which may influence how the information should be represented. Additionally, some information needs have pertinent temporal properties, from which past states can be aggregated and compared with the current state (e.g., coverage deltas, historical test performance).

While working on their projects, developers create and access a wide variety of external information sources. While these sources vary by development team and project, we have included a non-exhaustive list of common software development activities identified from prior work that often require external information; these have been categorized below by the developer's goal when accessing the information.

Crucially, many of the information sources used to answer these questions are stored externally to the project itself and is accessed using different tools, online services, and result files generated as byproducts of the development process. Each of these is external to the resource the developer needs to modify to accomplish their ask: the source code itself.

a) Team Awareness: Since all large software projects are built by teams, developers rely on awareness information to support collaboration and to keep appraised of their teammates activities [15]. Being unaware of their teammate's actions can negatively affect developers' own performance [16]. The kinds of questions developers might ask surrounding awareness include *"Who is working on what?"* and *"Who changed this [code]?"* [3]. To answer these questions developers often access the version control system to check on recent changes to the code they are working on. One commercial tool that supports this kind of activity is *vscode-gitlens* [2] which integrates some version control (specifically Git) features directly into the VS Code development environment. This kind of tool enables developers to quickly learn who, why, and when a line of code was changed with inline annotations.

b) Code Comprehension: Before adding new features or improving existing code, developers often need to understand the code they are working with. Source code comprehension is a core software engineering activity [17]. Code comprehension tasks are often time consuming [18] and involve finding answers to questions like *"Where is this method called or type referenced?"* and *"What are the values of these arguments at runtime?"* [1].

Code comprehension tasks can also have temporal properties. These may include questions like *"What is the evolution of the code?"* and *"What caused this build to break?"* [3]. Change-related information includes change metadata (who and when a change was made), how the code was changed, and the impact code changes can cause. Developers sometimes seek to answer *"What is the intention behind this piece of source code?"* [19]. Recent work demonstrated a combination of approaches to tackle the concern location problem for linking source code to specific software features [20].

c) Test Analysis: Test cases represent a special kind of source code that has important properties for developers. Similar to code changes, metadata information such as authorship can help to find the developer responsible for an individual test cases (e.g., *"Who owns a test case?"* [3]). Additionally, developers might be interested in co-change analysis (e.g., *"What test cases frequently change when this source code is modified?"*).

Additionally, test cases have important temporally-ordered dynamic properties (e.g., *"When did this test last fail?"*, and *"Does this test frequently fail?"*). The information to answer these questions is infrequently stored directly with the source code and is instead often maintained in external continuous integration systems such as those provided by TravisCI[3] or Jenkins[4].

d) Navigation and Debugging: Developers frequently navigate through the source code in their projects to answer specific queries. External information can also assist with these tasks, helping developers quickly find locations where code should be changed or added [21]. Online services can also be used to answer important development questions, such as *"what are the churn rates for all Java projects"*, that would be challenging or impractical to answer using local analyses [22].

[2]https://marketplace.visualstudio.com/items?itemName=eamodio.gitlens
[3]https://travis-ci.org/
[4]https://www.jenkins.io

Analogously, debugging often involves navigating through the code to answer questions like *"Why is [variable] equal to [value]?"* [2] or *"What is average execution time of this statement?"* [5].

C. Presentation views

Modern code editors such as Atom[5] and VSCode[6] are highly customizable user interfaces using common technologies like HTML, CSS, and JavaScript. Large communities of developers have created code editor plugins for these editors. Examining a broad set of existing source code editor plugins, we noted that their most prominent presentation techniques can be split into two categories: those that adorn the code editor with inline views within the source code itself and those that present the information in isolated views independent from the source code. From our examination of existing plugins, both of these categories can be further decomposed into unique visual representations.

1) Inline views: These views have the benefit that they are anchored by the source code they adorn. Figure 1 shows five of the most common representations used in existing plugins.

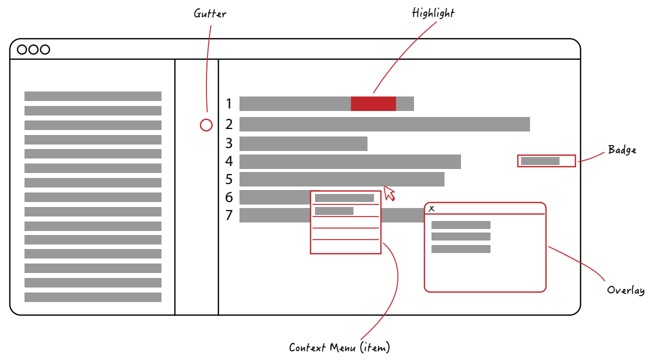


Fig. 1. Commonly used inline source code editor representations used for integrating external information. Of these, gutter (left margin) and badge (right margin) visualizations are the most commonly used.

Each of these representations has benefits and drawbacks:
Gutter: Most source code editors have a reserved presentation space to the left of each editor line called the gutter. Gutter representations place an icon or label in the gutter to convey a small piece of information about a line of code, function/method, or class. Gutter representations can typically accommodate small charts, single words, or icons. Due to space constraints, usually only a single adornment can be used for each line of source code.
Badges: Badges appear to the right of source code lines. Unlike gutter adornments, badges typically appear *within* the source code line. As most source code lines are not of uniform width and usually do not take the full horizontal viewport, badges offer greater space for providing more expressive information than gutter representations. Badges can appear as text, icons, small graphics, or a combination of these as

[5]https://atom.io/
[6]https://code.visualstudio.com/

appropriate. Multiple badges can appear together provided enough space, although when a line of code uses the full viewport these may be hidden.

Highlight: Highlighting portions of a line of code (like a variable or method invocation) can help provide additional information about the highlighted element, for example by highlighting other similar variable usages or method calls. Hovering a pointer over the highlighted portion can also provide additional information for that region in the form of a short tool-tip (text description). It is normally not possible to place more than one highlight on a single portion of source code, although a single line can have multiple highlights as long as they do not overlap. Unlike gutter or badge representations that can often convey their meaning just by glancing at them, code highlights normally need to be explicitly interacted with (e.g, by activating or hovering over them) to convey their meaning, although code colouring schemes are a special kind of code highlighting that does not require interaction as they are always activated.

Context menu items: Context menus are often used to provide access to information that might be less commonly used, be of lower priority, or could be overwhelming if it were always displayed. In these cases, the information can be associated with the code element (as with highlighting above) but without any visual representation except when invoked using the context menu in the code editor. By right clicking on the code element, the developer can be shown additional information about the code element. A primary downside of these representations is that it can be unclear which code elements have additional context menu options added to them which can inhibit their discoverability. Additionally, developers need to remember that the context menu option exists and remember to invoke it when needed.

Overlay: These representations are often used to augment the four other representations above. Overlays are panels that appear over code elements when the developer clicks on another representation (e.g., a gutter, badge, highlight, or invokes a context menu option). Overlays are often used to provide a more complex visual representation than is possible with the other less intrusive representations. This provides the developer the control to only display those overlays for which they need additional information.

2) Isolated views: These views present information within the development environment, but outside of the source code itself. Isolated views are often helpful when a developer is trying to get an overview or summary of the source code and is not sure which segment of source code they should be examining. Figure 2 shows common isolated mechanisms used within IDEs. For example, when trying to improve coverage, a developer might not know which file contains the highest percentage of uncovered lines. A ranked list of source files can deliver this information, but this list only exists in a full coverage report, and is not suitable to put inside the code area for a specific element as the developer would then need to scan all code elements to check their coverage levels. Figure 2 shows the two most common isolated information views used within code editing environments; as with inline views these also have strengths and weaknesses:

Notification item: Most development environments have a status bar that can be used to notify a developer of different events. An icon can be placed in this notification space to let the developer know that there is additional external information available. Different icons can be used for different notifications, or a single notification icon can reveal a list of individual actions if more than one are available. To avoid distraction, notification icons are generally subtle and are not intrusive, although this could increase the likelihood that developers miss an important notification if it could have helped the developer with their task.

Notification panel: By far the most common isolated representation is the notification panel; this modality is used extensively by plugins that augment IDEs. Notification panels are displayed in their own view adjacent the source code editor to provide additional information. These panels usually provide some kind of an overview that gathers individual pieces of inline information about specific code elements into a summary view in some kind of organized manner (e.g., a hierarchically sorted tree), although may use their own novel visualization mechanism. Since notification panels tend to be large, they must compete with each other for screen space. This means they are often turned off (or just hidden in non-activated tabs), forcing the developer to remember both that the tab exists, and that they should activate it, when they are performing aspects of a task the panel could assist with.

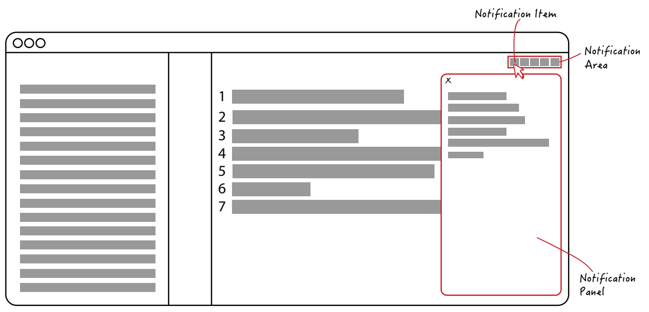


Fig. 2. Commonly used isolated visualizations for integrating external information into source code editors.

D. Leveraging presentation views

When determining how to integrate an external information source into an IDE, developers need to balance the actionability of the information with the overwhelmingness of the visual representation. While an external information source could be represented using several different presentation views, it is important that the designer consider the design principles in Section II-A when making their design choices.

This is especially important in an information-dense environment like the IDE as it is easy to imagine representations that would be overwhelming and intrusive given the breadth of information sources available in large mature projects. While embedding external information into source code editors can increase the accessibility of the information (as the developer does not need to access the external source to see the information), it must still be functional and must co-exist with other information sources.

For example, an online test coverage service might use gutters for each source code line to indicate whether that line of code is covered. When the developer clicks a gutter, an overlay can popup showing the tests that cover the line. An isolated presentation view could also be shown with a notification item can be displayed to indicate overall coverage condition (e.g., relative to a project-specific threshold) and when clicked a notification panel would appear with a full coverage report of all files in the project. At the same time, adding a gutter icon for every line of code may be overwhelming for the developer and would prevent other information sources from adorning the gutter (because there is only space for one element); in this case, using a badge for method-level coverage might be more helpful and allow the developer to expand the badge to access more detailed reports about problematic methods.

To better understand how developers perceive the utility and intrusiveness of the different visualizations, we sought to gather their feedback in the form of an in depth survey examining concrete external information integrations for a variety of common development tasks.

III. Integration Representation Survey

To understand developers' preferences for the visual representations of external information we conducted a survey and distributed it online among software developers.

While examining existing plugins that integrate into development environments, we noted that isolated representations (especially custom panels) were the most common form of visual integration; we wanted to determine if this matched with developer preferences. Additionally, given the flexibility and ubiquity of gutter and badge visualizations among inline representations, we wanted to further understand developer preferences among these options.

Consequently, the goal of the survey was to answer two main research questions:

RQ1 Do developers have preferences among inline or isolated visual representations?

RQ2 For inline representations, do developers prefer gutter-based or badge-based visualizations?

A. Survey methodology

The survey was designed to take 10 minutes and was distributed via email to our academic and industrial contacts and on social media through developer-relevant forums on Reddit and HackerNews. Each respondent was provided with four common tasks they might perform using external information. For each task, they were given two visual representations that we created by creating prototypes for an existing IDE.

In addition to selecting between their preferred representation for each task, the survey solicited rationale from the participants for their selection. Ultimately 29 respondents started the survey and 21 completely all four tasks (72.4%). Unfortunately, we do not know how many people saw our online recruitment and cannot calculate a response rate. 95.7% of respondents self-identified as professional developers and an average of 8.1 years of development experience.

The four tasks were selected from the author's experience using external information sources. While the tasks are clearly not exhaustive of all possible development tasks, we believe they represent a reasonably diverse cross section of development tasks that could leverage external information. The four tasks were provided in a randomized order, as were the two design alternatives for each of the tasks. The specific questions posed for each task were:

HIST This task required examining the past history of test failures as recorded by the Continuous Integration service which is executed as an online platform. *"You are facing some test case failures in a well-tested project. To understand the importance of these failures, you are trying to understand the historical behavior of failing tests."*

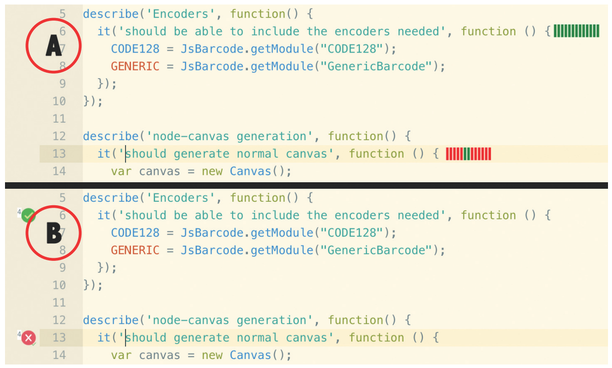
TEST In this task, the developer wanted to verify that their changes would not cause the tests to fail when executed with the regression test suite. *"You are working on a project with version control and continuous integration. After you make a commit, or you pull commits, you want to know whether some change breaks the build or fails some tests."*

PROF The profile task involved trying to understand what portion of code is slow by examining the online performance test suite which is run in a controlled environment to ensure that the results are consistent for all developers. *"While working on a source file in your project, you are trying to figure out what part of the code is executing slowly."*

COVER The coverage task involved examining the test coverage results as collected by the Continuous Integration service and hosted by the coverage service. *"While working on a source file in your project, you want to know which lines are covered by test cases and the current status of those tests."*

The visual representations for these four tasks provided badge-based and gutter-based representations. For example, for the HIST task, Figure 3 was shown. For each task, respondents were able to select between several rationale related to why they chose one representation over another, or were able to provide their own rationale if they had a different reason. After choosing between badges and gutters, they were also shown an independent representation that used a separate panel that did not try to fit the external information inline and were given the choice to choose between these. For example, for the HIST task the separate panel shown can be found in Figure 4. Further screenshots for these visual representations is available in the M.Sc. thesis[7].

[7]https://hdl.handle.net/2429/71778



```
describe('Encoders', function() {
  it('should be able to include the encoders needed', function () {
    CODE128 = JsBarcode.getModule("CODE128");
    GENERIC = JsBarcode.getModule("GenericBarcode");
  });
});

describe('node-canvas generation', function() {
  it('should generate normal canvas', function () {
    var canvas = new Canvas();
```

```
describe('Encoders', function() {
  it('should be able to include the encoders needed', function () {
    CODE128 = JsBarcode.getModule("CODE128");
    GENERIC = JsBarcode.getModule("GenericBarcode");
  });
});

describe('node-canvas generation', function() {
  it('should generate normal canvas', function () {
    var canvas = new Canvas();
```

Fig. 3. Inline representations for badges (A) and gutters (B) for HIST.

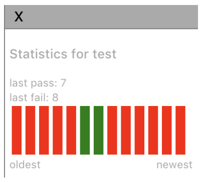


Fig. 4. Isolated representation for the HIST task using its own panel.

B. RQ1: Inline vs. isolated representations

One drawback with supporting integration of external information is that the information could distract developers. This is especially true if one information source monopolizes the source code area or multiple external information sources are trying to decorate the same portion of source code. One advantage of allowing the information to be integrated is that it could increase the utility of the information (both by being more discoverable and by allowing direct usage). One advantage of allowing the information to be hidden and only displayed when toggled by the developer is that the information could be made less distracting to them as they work.

Figure 5 shows developer preferences between showing information in an isolated panel or inline with code. For all scenarios, most respondents prefer to have the information shown in the code area (inline). Several respondents did mention though that having an optional summary panel could also be useful in addition to the inline representation.

1) Inline vs. toggled representations: Respondents who preferred inline representations mainly rationalized their preference for reasons related to **minimalism**. Unobtrusive representations gave them the option to have access to the information without it overwhelming their code editors:

> "*Stay minimal until I ask for more*" — P4 HIST
> "*Keep it minimal and inline maybe like the ones offered in VSCode*" — P18 HIST
> "*Stay minimal until I ask for more information*" — P4 TEST
> "*I prefer inline with the option to explore in an extra panel*" — P18 PROF

For those respondents who preferred separate panels, their primary rationale was because it enabled them to **only consider additional information when it is needed**:

> "*Ability to hide the information when not required.*" — P2 HIST
> "*Ability to show the additional data when required.*" — P2 TEST
> "*Ability to hide information until required*" — P2 COVER

Additionally, respondents had specific scenarios where separate panels had **additional capabilities** that inline presentations do not have:

> "*Extra panel can be sorted*" — P25 PROF
> "*It may be easier to jump across files with the panel. The bottom part of the panel also helps*" — P26 COVER
> "*Extra panel could in theory show extra information about each relevant commit in the bar chart.*" — P6 HIST

That said, many respondents also specifically mentioned that there were too many panels in their editors already, and that for some tasks the inline representation was simply a better match for the kind of task the data would be used for (for example *"inline with the editor panel makes for faster REPL workflow"* – P6 COVER.

2) Combined representations: A few participants want to be able to combine both inline and independent representations because they liked to have options to match their needs, for example "*Nice to have options*" — P1 PROF. This sentiment was echoed by several of the respondents who preferred the inline representations but sometimes found the separate panels handy. This was mainly due to the separate panel's ability to summarize information in a hierarchical fashion, although sometimes having both allowed tackling problems in both top-down and bottom-up ways: "*[the panel] shows whole project coverage by file, while the gutter shows coverage by line.*"

RQ1: Should representations be inline or isolated? Inline representations were considerably more popular with the surveyed developers because they do not like the proliferation of panels around their code editors. Simultaneously, developers found isolated panels better at presenting summary data and have some capabilities inline representations lack. The option to activate an isolated panel from an inline representation could increase the utility of both representations.

C. RQ2: Inline representation preferences

For two of the tasks, developers preferred badges over gutter visualizations (HIST: 65.2%, PROFILE: 73.9%). For the other two tasks badges were not preferred (TEST: 45.8%, COVER: 27.3%). While variation between tasks suggests that the characteristics of the task remain an important factor when choosing between these representations, the qualitative

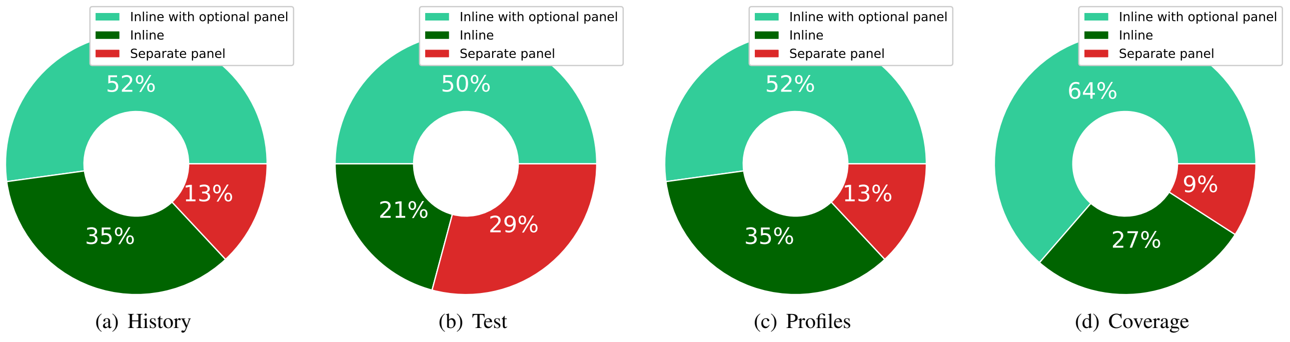


Fig. 5. Preferences for showing information in a separate panel for each scenario.

comments did seem to indicate that badges had more positive attributes.

For respondents who preferred badges over gutters, their primary rationale was that *"A provides more information at a glance and saves me from the extra step"* (HIST: 93.3%, TEST: 81.8%, PROFILE: 76.5%, COVER: 83.3%). This suggests that respondents chose badges mostly because of **the higher information density**. Concerns about using the gutter area for external information seemed to be task dependent, *"[The gutter] occupies gutter area, which is used for setting breakpoints"* (HIST 13.3%, TEST 27.3%, PROFILE 41.2%, COVER: 50.0% The primary challenge identified with developers with respect to gutter-based representations reflect readability challenges with these limited representations:

> "*[The badge] is visually easier to read.*" — P2 TEST
> "*'Failed' text is hard to read.*" — P11 PROF
> "*The gutter icon seems hard to read.*" — P28 PROF

For respondents who preferred gutters over badges, their primary rationale for selecting gutters were that badges "*are in the same space as the source code*" (39%), "*the gutter is easy to spot*" (48%), and gutters "*provide enough information... while keeping visually minimal*" (55%).

For these respondents, their rationale mainly centered around minimizing visual distraction:

> "*when you test hundreds of time, the bars area might get too long.*" — P7 TEST
> "*I like the icon in [the gutter].*" — P27 PROF
> "*this gutter info is clear and easy to read. Also, [...] there is a lot more inline text in A that can really clutter up the text area.*" — P11 COVER

The variation in opinions suggests that integrators of external information need to carefully consider the representation that will work best for the information they are trying to show. One reason for this is that the information density developers need varies. In the coverage task a binary piece of information was needed (whether the element is covered or not) while in the profiling task more descriptive information is required.

One distinct disadvantage inline representations have compared to independent panels occurs when multiple external information sources try to decorate the same code location. While each independent panel can provide isolated views that do not need to interact with one another, inline representations can interfere with each other. To investigate how these can be handled, we provided respondents with visual strategies for resolving collisions in badge and gutter representations and asked them to comment on these.

For this question, 76% of respondents preferred the badge-based collision representations. Their rationale mainly stemmed from the ability of badges to convey more information making it easier to differentiate what was being shown:

> "*It is easier to read on the right side.*" — P11
> "*More information is displayed (and in a pleasing manner) making it easier to draw accurate conclusions.*" — P19
> "*[Badges are] more intuitive and prominent. It does not feel annoying despite always there.*" — P21

For the 24% of respondents who prefer gutters, their sentiments were mainly related to information overload:

> "*[Gutters] feel less visually jarring; there's only one line that expands horizontally. [Gutters] also avoid an "explosion" of information and reminds me of a context menu (feels more familiar).*" — P20
> "*[Badges] really seem like an overload of information and would hardly fit anymore if code lines are long and take up a big part of the screen.*" — P28

RQ2: Do developers prefer gutters or badges? For the four surveyed tasks, badge-based visualizations were slightly preferred by developers over gutter-based visualizations. The ability of badges to convey more information seems to be the primary driver of this preference. Correspondingly, space constraints imposed by gutters can cause in readability issues. Gutter visualizations work for tasks where information can be easily and naturally fit a small footprint. When more information is needed, or multiple pieces of information vie for the same inline locations, badges are preferred.

IV. EICE Technology Probe

While the survey in Section III helped us to understand developer preferences, to gather further insight into developer performance we created the External Information in Code Editor (EICE) prototype. This prototype was designed to help us evaluate how developers would interact with external information visualized within their code editors while performing real tasks. In particular, we wanted to ensure that the visualization of external information within the developer's environment did not overwhelm them and negatively impact their performance. EICE was developed as a plugin for the Atom development environment.[8]

A. Controlled experiment

While the online survey focused on visual preferences for integrating external information, we wanted to gain a better understanding of how developers could actually use this information using the EICE technology probe. To do this, we performed a controlled experiment which set out to answer two primary research questions:

RQ3 Does integrating external information impact task performance?

RQ4 Are there negative consequences to integrating external information?

a) Participants: At the conclusion of our prior survey, participants self-identified as being amenable for future contact. From this pool 8 survey respondents became study participants, each had at least one year of JavaScript and an average of 7.5 years of professional development experience.

b) Project: To select a real project for the study, we searched GitHub for projects written in JavaScript, with at least one year of development history with between 20 and 40 files, and over 200 commits from 10 developers. We excluded projects that did not have unit tests and that we could not easily get to build. From these, we randomly selected the JSBarcode project[9]. JsBarcode consists of 615 commits by 24 developers over a 7 year period at the time of the study.

c) Tasks: The experiment involved four tasks (T1–T4). As with the tasks in Section III-A, these four tasks were selected from the author's experience; they were different than the prior four tasks to gain insight into developer feedback for different tasks requiring external information. Each task was independent of each other and used a different external information source; external information was dynamically loaded as the participant worked. The four experimental tasks were:

T1 *Comprehension (Runtime value):* For values covered by unit tests, EICE will display the value before and after the line executes. Developers interact with an overlay representation by clicking on a variable and requesting runtime values.

T2 *Build (Broken build):* EICE adds gutter items to lines that belongs to commits that directly or indirectly cause the build to fail. When users click the gutter item the commit SHA and related metadata will be displayed in an overlay window.

T3 *Test (Coverage):* EICE pulls coverage data from the latest coverage report and adds gutter items to lines captured in the report. Green gutter items indicate the lines are covered and how many times times they are covered. Red gutter items indicate the lines are not covered by unit tests.

T4 *Optimization (Profiling):* EICE extracts profiling results of functions and shows a badge next to each function's declaration, showing how long it takes to run along with the runtime of the entire program.

[8]Implementation details in M.Sc. thesis: https://hdl.handle.net/2429/71778
[9]https://github.com/lindell/JsBarcode

Participants did not need to write any source code, but were asked to diagnose and propose a fix for each problem after investigating the JSBarcode source code and associated resources. The approximate time to complete a task was estimated to be 10 minutes. Combined with the study introduction and follow-up, the total duration of the study was one hour.

Each participant performed four tasks in a randomized counterbalanced design. Two tasks were performed with the EICE technology probe while two tasks were performed with the control treatment where participants would use tools of their choosing for accessing external information.

d) Metrics: Given the modest number of participants in the experiment, and to gain as much insight as we could in a limited controlled setting, we designed the experiment to focus on the *number of actions* **NA** and the *number of resource switches* **NRS** rather than the time required to complete the task. The intuition behind these measures is that the more actions a developer needs to take to complete a task, the more effort the task may require. Additionally the more resource switches and programs involved in completing the task, the more complex the process becomes, increasing the cognitive load to complete the task.

We instrumented the experimental computer to automatically count a resource switch whenever the participant switched from one application to another application. Resource switches were also counted whenever the developer switched between files in the code editor, web pages in the browser, or tabs in the editor or browser.

While these measures are attempting to gain some insight into relative developer performance, the high level goal of the study was really to gather evidence that EICE did not unduly burden developers as they worked; these measures are only meant to be proxies to gain lightweight insight into the impact of the tool.

B. RQ3: Impact on task performance

The results for the number of resource switches (NRS) for the experimental and baseline tasks showed that participants using EICE performed 33% as many resource switches (14 switches with EICE compared to 32 switches without EICE). In fact, for three of the four tasks (T1, T3, T4), EICE participants were able to use the external information embedded in the code editor alone to complete the task. One of the participants reflected:

> "*If I'm already working on something I don't need to worry about switching to all different applications.*"— P4

In terms of the number of actions (NA), as with NRS, participants using EICE for T1, T3, and T4 performed better than with their traditional tools; across all tasks participants using EICE performed 71% as many actions. This decreased impact (relative to NRS) was largely related to T2, for which participants found EICE hard to use; across the other three tasks, EICE required only 51% as many actions to complete the tasks.

Ultimately though, the goal of this small controlled experiment was not to quantify the effort reduction of using EICE vs traditional tooling. We mainly wanted to evaluate, with real tasks using real external information from a real project, that the visual integration did not overwhelm or confused developers causing them to to perform worse than with their traditional tooling. In this way, we believe this experiment provides initial hints that such integrated information can be both useful and be visualized in a way that is not overwhelming to developers.

The participants also reflected positively on how having access to this external information could impact them on their own tasks. For example, " *[Even] I've disabled communications, and I'm working, but I still see that a really relevant information because it would be showing me in the code that I'm working on. So that can be quite helpful.*" — (P4)

RQ3: Does integrating information impact performance? Participants were able to successfully use external information embedded into their source code editor to both decrease the number of resource switches and number of actions for most of the experimental tasks.

C. RQ4: Negative consequences of integration

Integrating external information does not automatically make developer's lives easier. For example, in T2 (broken build) the EICE group performed worse than the control group in terms of both NRS and NA. This was due to a design flaw in how we integrated the build breakage external information into the code editor: a usability decision we made (to open the failure report when the participants clicked on the inline visualization) caused participants expected to perform extra resource switches and extra actions compared to the control group. This demonstrates the importance of careful planning and usability testing for these kinds of visualizations as they can actually decrease developer performance.

Reflecting on their own projects, developers still had lingering concerns about the integration becoming distracting: "*[the inline representation] might be too distracting in some cases*" — (P3), or that they would like to have a toggle to easily turn off information sources they were not interested in: "*having the option to toggle on and off [the visual elements]*" — (P1).

RQ4: Can inline integration be detrimental? When integrating information into the source code editor, designers must carefully consider how developers will use and interact with the information to ensure they do not actually cause the developers to perform more work.

V. Discussion

Developers use both code editing environments and external development tools on a daily basis. This paper has provided a glimpse into developer preferences for how code editors can be augmented with external information. While the results may seem straightforward, we believe they can provide concrete design guidance for future tool developers. Specifically, developer guidance from the survey suggests that increased usage of badge-based representations can both allow tool developers to use more verbose representations (e.g., compact textual summaries or graphs). Developers also found these representations easier to use when multiple information sources are trying to decorate the same code artefacts. Feedback from developers also showed that in addition to integrating this information into their code editors, they also appreciated direct linking from the inline representation back to the original information source so they could see it in its usual context if they needed additional information.

Additionally, results form the experiment suggest that even straightforwardly integrating existing information from external sources into the code editor can provide meaningful value for developers. While the experiment primarily focused on developer's ability to use the integrated information, we believe surfacing the information and making it accessible without overt effort is just as important. These results have implications for the importance of surfacing information and the amount of immediate context needed for that exploratory start point. By integrating external information into the code editor, developers are relieved from not only the search step (knowing where to find a piece of information for a specific cod artefact), but also from the ideation step where they need to *think* about searching for the information. Given the value these links can provide to developers, additional research on mechanisms to support exposing information from online information sources may be warranted (to complement existing research projects that infer these links).

A. Threats to validity

There are several threats to validity for an exploratory study such as this one. These are important concerns that must be taken into consideration when framing the results presented in this work which is providing initial evidence for the utility and preferences for the visualization of external information within development environments.

a) Internal validity: The dynamic structure of the survey made it challenging to directly compare all participants as the followup questions were based on their prior responses (e.g., the rationale for choosing A or B). This decision was made to increase the feedback we could get from respondents, but injected some of our own design thinking into the survey process.

For the controlled experiment, we did not examine participants as they worked on their own projects and tasks. This would have allowed us to more directly examine the utility of the information rather than using proxy metrics like number of actions / context switches, but would have eliminated our ability to have a meaningful control group without a much larger participant pool.

b) External validity: The primary threat to external validity is the participant sample sizes of the survey and experiment. While this is a clear limitation, the goal of the work was to provide insight into developer preferences (survey) and initial evidence whether integration decreased performance (experiment). Additionally, the number of tasks evaluated in the survey and the reliance on a single project for the experiment both limit the generalizability of the results. As with the primary threats to internal validity, an *in vivo* study would be the next step to gain additional insight into the challenges and benefits of integrating external information into code editors in practice.

VI. Related Work

Prior work has explored various methods to deliver external information to developers including (1) software visualization, (2) tools adding separate displays to code editors, and (3) through novel tools that deliver their results using inline code editor representations. This work differs from many prior efforts in that it focuses on developer perceptions of the representations themselves, rather than prior work which are predominantly novel tools that happen to surface their results within the code editor. A non-exhaustive overview of this work is given below.

A. Software visualization

Software visualization provides another perspective to aid code understanding. Prior work such as Software Landscapes [23], Software World [24], Component City [25], and CodeCity [26], [27] all visualizes software allowing for data exploration and analysis. Such approaches typically replace code editor views with novel and independent visual representations. In contrast, in this work we investigated developer sentiment towards approaches that enable developers to not leave their familiar code editing environment.

B. Code editor replacements

There are previous studies that attempt to adding "side views" to code editors to presenting external information. Reacher, an Eclipse plug-in, shows the call graph of methods of interest next to the code editor view [28]. Software terrain maps, based on the metaphor of cartographic maps, provides an additional visualization display for developers to navigate around the source code in the editor [29]. Code Bubbles [30], Code Canvas [31] and Debug Canvas [32] take on a different approach: they create a larger view where source code views are enclosed in bubbles or regions that can be independently laid out on a canvas. These representations can further display relations among these smaller regions, easing the code comprehension and debugging process. Despite the perceived value in such interfaces, they introduce new scalability issues that are not as problematic for our less integrated approach [30].

C. Code editor augmentation

Tools such as HATARI [13] and GitLens[10] CodeMetrics[11]. Whyline [2] and PerformanceHat [5] integrate external information by adding visual changes (e.g., annotations) directly to the source code, without (necessarily) providing a separate independent display. HATARI plug-in for Eclipse adds annotations to source code elements, using colors to indicate risk levels of code changing [13]. GitLens uses annotations including highlight and heat-map to seamlessly integrate Git core features in the code editor itself. CodeMetrics shows the complexity information above the function declarations. Whyline is a tool developed recently to allow developers to select questions about a program's output and work backwards from output to its causes [2]. PerformanceHat creates operational awareness of performance problems and integrates runtime performance traces into source codes, displaying the performance statistics/impacts in the form of overlays [5]. This type of tool is closest to the EICE technology probe in that developers do not have to leave the code editor to benefit from the tool's results.

VII. Conclusions

In this paper, we investigate whether developers are open to greater integration of the data from external information sources into their development environments. In particular, we wanted to learn more about developer preferences for the visualization of this information and to check whether they found it overwhelming or improved their ability to perform some of their development tasks.

We did this through an online survey with 21 responses which suggests that developers prefer the additional information being visualised within their environments as long as it is carefully done. In particular, developers prefer inline representations that place information before the code (as gutter icons) or after the code (as badge representations) over representations that use isolated panels or views. A common concern among developers is that they do not want to be overwhelmed by the integrated information. Through a controlled experiment we show initial evidence that developers are able to leverage integrated information to both decrease the number of actions and context switches required to perform development tasks.

Ultimately, the survey and controlled experiment provide initial evidence that developers support visualizing external information in their environments and provides guidance for future tool developers to consider when designing these visualizations.

[10]VSCode plug-in for visualizing code authorship via Git blame annotations and code lenses. (https://github.com/eamodio/vscode-gitlens)

[11]VSCode plug-in for computing and displaying code complexity inline with functions. (https://github.com/kisstkondoros/codemetrics)

References

[1] J. Sillito, G. Murphy, and K. De Volder, “Asking and Answering Questions during a Programming Change Task,” *Transactions on Software Engineering (TSE)*, vol. 34, no. 4, pp. 434–451, 2008.

[2] A. J. Ko and B. A. Myers, “Finding causes of program output with the Java Whyline,” in *Proceedings of the International Conference on Human Factors in Computing Systems (CHI)*, 2009, pp. 1569–1578.

[3] T. Fritz and G. C. Murphy, “Using information fragments to answer the questions developers ask,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2010, pp. 175–184.

[4] E. Murphy-Hill, R. Jiresal, and G. C. Murphy, “Improving software developers’ fluency by recommending development environment commands,” in *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 1–11.

[5] J. Cito, P. Leitner, C. Bosshard, M. Knecht, G. Mazlami, and H. C. Gall, “PerformanceHat: Augmenting source code with runtime performance traces in the IDE,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2018, pp. 41–44.

[6] T. Zimmermann, “Changes and bugs mining and predicting development activities,” Ph.D. dissertation, Jan 2008.

[7] R. DeLine, G. Venolia, and K. Rowan, “Software development with code maps,” *Communications of the ACM (CACM)*, vol. 53, no. 8, p. 48–54, Aug. 2010.

[8] G. Venolia, “Bridges between silos: A microsoft research project,” January 2005. [Online]. Available: https://www.microsoft.com/en-us/research/publication/bridges-between-silos-a-microsoft-research-project/

[9] A. Begel, Y. P. Khoo, and T. Zimmermann, “Codebook: Discovering and exploiting relationships in software repositories,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2010, pp. 125–134.

[10] D. Spinellis, “Developing in the cloud,” *IEEE Software*, vol. 31, no. 2, pp. 41–43, Mar 2014.

[11] G. Venolia, “Textual allusions to artifacts in software-related repositories,” in *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, 2006, pp. 151–154.

[12] D. Čubranić and G. C. Murphy, “Hipikat: recommending pertinent software development artifacts,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2003, pp. 408–418.

[13] J. Śliwerski, T. Zimmermann, and A. Zeller, “HATARI: Raising risk awareness,” in *Proceedings of the European Software Engineering Conference held jointly with International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2005, p. 107–110.

[14] J. Singer, “Practices of software maintenance,” in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 1998, pp. 139–145.

[15] J. Chisan, “Towards a model of awareness support of software development in GSD,” in *Proceedings of the International Workshop on Global Software Development (GSD)*, 2004, pp. 28–33.

[16] H. C. Estler, M. Nordio, C. A. Furia, and B. Meyer, “Awareness and Merge Conflicts in Distributed Software Development,” in *International Conference on Global Software Engineering (GSE)*, 2014, pp. 26–35.

[17] M. P. O’Brien, “Software Comprehension – A Review & Research Direction,” p. 29.

[18] N. A. Al-Saiyd, “Source code comprehension analysis in software maintenance,” in *Proceedings of the International Conference on Computer and Communication Systems (ICCCS)*, 2017, pp. 1–5.

[19] A. J. Ko, R. DeLine, and G. Venolia, “Information needs in collocated software development teams,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2007, pp. 344–353.

[20] M. Eaddy, A. Aho, G. Antoniol, and Y.-G. Gueheneuc, “CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis,” in *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2008, pp. 53–62.

[21] S. Bajracharya, J. Ossher, and C. Lopes, “Sourcerer: An infrastructure for large-scale collection and analysis of open-source code,” *Science of Computer Programming (SCP)*, vol. 79, pp. 241–259, Jan. 2014.

[22] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, “Boa: A language and infrastructure for analyzing ultra-large-scale software repositories,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2013, pp. 422–431.

[23] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz, “Software landscapes: Visualizing the structure of large software systems,” in *Proceedings of the Transactions on Visualization and Computer Graphics (TCVG)*, 2004.

[24] C. Knight and M. Munro, “Virtual but visible software,” in *Proceedings of the International Conference on Information Visualization (IV)*, 2000, pp. 198–205.

[25] S. M. Charters, C. Knight, N. Thomas, and M. Munro, “Visualisation for informed decision making; from code to components,” in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2002, pp. 765–772.

[26] R. Wettel and M. Lanza, “CodeCity: 3d visualization of large-scale software,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2008, pp. 921–922.

[27] R. Wettel, “Scripting 3d visualizations with codecity,” in *Proceedings of the Workshop on FAMIX and Moose in Reengineering (FAMOOS)*, 2008.

[28] T. D. LaToza and B. A. Myers, “Visualizing call graphs,” in *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2011, pp. 117–124.

[29] R. DeLine, “Staying Oriented with Software Terrain Maps,” in *Workshop on Visual Languages and Computation (VLC)*, 2005, pp. 309–314.

[30] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola Jr, “Code bubbles: Rethinking the user interface paradigm of integrated development environments,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2010, pp. 455–464.

[31] R. DeLine and K. Rowan, “Code Canvas: Zooming towards better development environments,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2010, p. 207.

[32] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss, “Debugger Canvas: Industrial experience with the code bubbles paradigm,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012, pp. 1064–1073.