Augmenting Code with In Situ Visualizations to Aid Program Understanding

Jane Hoffswell
University of Washington
jhoffs@cs.washington.edu

Arvind Satyanarayan
Stanford University
arvindsatya@cs.stanford.edu

Jeffrey Heer
University of Washington
jheer@cs.washington.edu

ABSTRACT

Programmers must draw explicit connections between their code and runtime state to properly assess the correctness of their programs. However, debugging tools often decouple the program state from the source code and require explicitly invoked views to bridge the rift between program editing and program understanding. To unobtrusively reveal runtime behavior during both normal execution and debugging, we contribute techniques for visualizing program variables directly within the source code. We describe a design space and placement criteria for embedded visualizations. We evaluate our in situ visualizations in an editor for the Vega visualization grammar. Compared to a baseline development environment, novice Vega users improve their overall task grade by about 2 points when using the in situ visualizations and exhibit significant positive effects on their self-reported speed and accuracy.

ACM Classification Keywords

H.5.2 Information Interfaces and Presentation (e.g., HCI): User Interfaces; D.2.3 Software Engineering: Coding Tools and Techniques

Author Keywords

Visualization; Code augmentation; Program behavior; Program understanding; Debugging

INTRODUCTION

Programmers perform a variety of high-level tasks during the software development process, including authoring, testing, debugging, and reviewing code [22]. Many popular development environments include specialized tools to support these tasks. For example, techniques such as logging, replay [5, 18, 28], and breakpointing interactively surface the internal state of the program, and are used to pinpoint specific lines of code responsible for the runtime behavior.

However, these existing techniques must be explicitly invoked and are presented to programmers in separate, coordinated views. Studies have shown that switching between these views imposes a burden on developers, making it difficult for them to maintain a clear picture of the overall context of the runtime behavior [22, 27, 29, 31]. This separation is particularly problematic when programmers are immersed within a particular task — they may overlook details that would be obvious with an alternative view [31]. Expectations about the desired behavior may additionally cause programmers to overlook errors when the behavior diverges from expectation.

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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CHI 2018, April 21–26, 2018, Montreal, QC, Canada

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-5620-6/18/04...$15.00
DOI: https://doi.org/10.1145/3173574.3174106

```
151 {
152   "type": "text",
153   "from": {"data": "drive"},
154   "properties": {
155     "enter": {
156       "x": {"scale": "x", "field": "miles" },
157       "y": {"scale": "y", "field": "gas" },
158       "dx": {"scale": "dx", "field": "side" },
159       "dy": {"scale": "dy", "field": "side" },
160       "fill": {"scale": "c", "signal": "mouseover_year" },
161       "text": {"field": "year" },
162       "align": {"scale": "align", "field": "side" },
163       "baseline": {"scale": "base", "field": "side" }
164     }
165   }
166 }
```

Figure 1. Code augmentations visualize the runtime state of program variables in a Vega [32] specification. A *histogram* shows the distribution of variables containing *set* data. Interacting with the `year` histogram filters all other histograms to only show the data values where the `year` is between 1995 and 2002 (hand cursor shown here for clarity; in a real implementation the cursor is hidden mid-interaction to aid chart reading). For `mouseover_year`, a *tick* visualization depicts value changes.

We hypothesize that automatically surfacing contextually relevant information directly within a code editor can help programmers better understand runtime behavior. We contribute in situ visualizations of program behavior intended to narrow the gulf of evaluation for software development tasks. We present a *design space* of embedded visualizations for interactive applications that visualize time-varying variables. Snapshots of the program state highlight the exact value of scalar variables, or the underlying distributions of set variables. We further contribute criteria for the *placement* of code augmentations based on trade-off metrics. Interactive visualizations of the program state can enable richer interactions within the development environment and present runtime information as a first-class component of the code authoring process.

We evaluate our in situ visualizations in an editor for the Vega visualization grammar [32]. In an evaluation with 18 novice Vega users, we found that participants improved their overall task grade by about 2 points when using the in situ visualizations. When reflecting on their experience, participants showed significant positive effects on their self-reported speed and accuracy. Participants found the visualizations particularly useful for understanding the dynamic behavior of the code.

RELATED WORK

This research is motivated by three areas of related work: empirical studies of programmers, program visualization and debugging, and text and environment augmentation.

Empirical Studies of Programmers

Many debugging tools utilize separate coordinated views, which require programmers to intentionally switch between code authoring and debugging tasks. However, this required shift of the programmer's focus can incur a high context switching cost and interrupt the programmer's flow.

Flow [27] is a well-studied topic from the field of psychology and describes the state in which an individual "operates at full capacity." While "in flow," programmers may introduce but overlook errors in their code. Saff & Ernst [31] describe a model of developer behavior where the phase in which programmers have unknowingly introduced an error is called "ignorance." Across two development projects, Saff & Ernst [31] found that Java programmers were ignorant of program errors for about 17 minutes on average (and sometimes more than 90 minutes). Identifying these errors requires programmers to switch from code authoring to debugging; Ko et al. [22] found that programmers spend 5% of their development time switching between tasks on average. We hypothesize that by visualizing program state in situ, programmers will be more aware of the impact of code changes and thus reduce the time spent ignorant of errors or switching between tasks.

Once an error has been introduced, programmers must invest time to find and fix the error. Ko et al. [21] found that programmers spend 46% of their time debugging. Saff & Ernst [31] found a significant relationship between the "ignorance time" of an error and the amount of time needed to fix it. Parnin & Rugaber [29] studied the time required to resume programming after an interruption and found that 30% of programming sessions had a lag of over thirty minutes between entering the session and authoring new code, potentially as a result of extensive debugging tasks. By surfacing program state earlier in the process via in situ visualization, we hope to reduce the time span between introducing and fixing errors.

Program Visualization and Debugging

Program visualization can facilitate both educational and debugging tasks. The Online Python Tutor [14] visualizes objects, variables, and stack frames allowing students to inspect the runtime state of their code. The Online Python Tutor has been used by over 3.5 million people and has been extended to support additional languages including JavaScript and C. Algorithm visualizations [4, 8, 33] can illustrate the behavior by visualizing each step in the algorithm and have been shown to improve understanding of the behavior [13].

Many debugging tools provide coordinated views to display relevant system information and facilitate tracing of the execution history. The Whyline [20] visualizes the path of runtime actions relevant to a "why" or "why not" question about the runtime behavior. Timelapse [5] visualizes web event streams and displays linked views of internal state information; breakpoints allow programmers to trace state information to particular parts of the original source code. FireCrystal [28] emphasizes the connection between code and runtime behavior by extracting the relevant CSS, HTML, and JavaScript code responsible for behaviors on a web page. Hoffswell et al. [18] describe visual debugging techniques that show the history of interaction events and relationships within the code to debug interactive visualizations. Our system provides a more direct link between program state and source code by directly augmenting a code editor with visualizations of program behavior.

Theseus [23] similarly narrows the connection between runtime behavior and code by displaying visualizations of program calls alongside the source code and call stack. In situ visualizations of program behavior have been developed for a variety of programming topics including code properties such as the edit history or code author [15], variable read/write accesses [1], performance behavior [2], and real-time programming tasks [34]. Bret Victor [36] describes design considerations for a programming system to support program understanding tasks aided by both code annotations and visualizations. Our work contributes to this body of work by presenting design considerations for the incorporation and placement of generic augmentations within code. We further describe the design space for visualizations of program state and temporal changes to program variables.

Text and Environment Augmentation

Text augmentations display supplemental information to support or extend the text [6, 38]. Tufte [35] identified sparklines as a way to incorporate small, data-rich visualizations into text. Goffin et al. [9] have generalized this idea to include any form of word-scale graphics. Goffin et al. [10, 11] additionally present design considerations for the placement of word-scale graphics and for incorporating interactions.Willett et al.'s scented widgets [37] augment standard navigational widgets with visualizations of page visitation or content engagement. Such visualizations help orient users within the space of content and help them to engage with underutilized content. Our work similarly aims to attract user attention to areas of interest by providing contextually relevant visualizations of program state that highlight interesting trends in the runtime behavior.

USAGE SCENARIO

Programmers often have a set of implicit assumptions about how their code will behave, which reflect their original intentions when writing the code. Consider a case in which a programmer is creating a scatter plot visualization that supports panning and zooming. The programmer may define `xMin`, `xMax`, `yMin`, and `yMax` values that track the view port position and update based on how much the user has dragged. These values can then be used to determine the current domain for each axis (`xDomain` and `yDomain`). The programmer decides to vary the point `size` based on the relative zoom level, using the span of the `xDomain` as a proxy. However, the programmer accidentally introduces an error where the `xMin` and `xMax` values are mutually dependent and therefore cause the `xDomain` to stretch as the user pans the visualization. With the code wired up to use all the variables, the programmer may begin testing the output via interaction.

The programmer starts by performing pan operations to see how the visualization updates based on the changing values of the axis min and max (`xMin`, `xMax`, `yMin`, `yMax`), the computed domains (`xDomain` and `yDomain`), and the point `size`. However, even while *only* panning, the scatterplot visualization seems to *also* zoom into the points. This behavior is surprising, as it does not reflect the programmer's intentions. Looking at the dynamic code behavior indicated by the inline visualizations, the programmer notices that the `size` is increasing while panning. The size is based on the span of the `xDomain` (which should not change while panning), revealing an underlying error in how `xDomain` is computed: the `xMin` and `xMax` values are mutually dependent and thus produce an error when updated sequentially.

DESIGN CONSIDERATIONS

Code augmentations narrow the gulf of evaluation between the programmer's code and the runtime behavior by surfacing contextually relevant information in situ. In this section, we describe three requirements that inform the design of effective in situ visualizations: they must be **comparable**, **salient**, and **unobtrusive**. These properties are motivated by prior work surrounding text augmentation and program visualizations.

In situ visualizations help programmers draw connections between variables in the code and their runtime behavior. Code augmentations should facilitate identification of important trends that can amend the programmer's understanding of the runtime behavior. **Comparability** is particularly essential for understanding the overall behavior. Programmers need to interact with multiple visualizations to understand the behavior of different variables in the code. For example, on-demand linking [3] can enable the programmer to understand the relationships between different variables. When comparability is essential to the programmer's current task (e.g., to compare the values of related variables), the augmentations should prioritize placement decisions that facilitate comparison across separate augmentations using alignment and shared axes [30].

The code augmentations should adapt to provide contextually relevant information for the programmer's current task and should update their **salience** to attract the programmer's attention to potential areas of interest. The placement [10, 38] and animation [16, 25] for a code augmentation influence the salience. The *temporality* (snapshot or sequence) of the augmentation impacts its utility for particular tasks.

The code augmentations must remain **unobtrusive** so they do not detract from the programmer's primary task. The amount of text *reflow* and *occlusion* [10, 38] can increase the obtrusiveness of the code augmentations. The code augmentations should minimize changes to the position and visibility of the code when the programmer is actively reading the code. However, violating such layout concerns may better maintain the code structure or improve augmentation salience at the programmer's periphery (e.g., while testing the program output).

DESIGN SPACE OF CODE-EMBEDDED VISUALIZATIONS

In this section, we describe the design space of embedded visualizations (Figure 2). Programmers must understand the values of, and changes to, variables as their code executes. Thus, we decompose the design space into two types of visualized data (value and set) at two levels of detail (data and change) across two temporalities (snapshot and sequence). We also consider issues of interaction, size, and placement.

Detail	Type	Temporality	Name	Visualization
Data	Value	Snapshot	Exact Value	{ "x": 42, ... }
		Sequence	Line	
			Horizon	
	Set	Snapshot	Heatmap	
			Histogram	
Change	Value	Snapshot	Indicator	*No Change:* *Change:*
		Sequence	Timeline	
			Tick	
	Set	Snapshot	Modification Indicator	
		Sequence	Stacked Area	

Figure 2. We identified ten visualizations for code augmentation based on the level of *detail*, variable data *type*, and *temporality* level of interest.

```
33 {
34   "name": "indexified_stocks",
35   "source": "stocks",
36   "transform": [{
37     "type": "lookup",
38     "on": "index", "onKey": "symbol" ,
39     "keys": ["symbol" ], "as": ["index_term" ],
40     "default": {"price": 0}
41   }, {
42     "type": "formula",
43     "field": "indexed_price" ,
44     "expr": "datum.index_term.price > 0 ? (datum.price - datum.index
45   }]
46 }
```

Figure 3. The "`index_term`" variable in this Vega [32] specification represents an object, so we select a representative property to visualize and differentiate the augmentation from others using color. On line 44, the programmer uses the `index_term.price`, so we choose "`price`" as the representative property. The selected key is also shown on mouseover.

Data Type: Value and Set

We separate program variables into two data types: value and set. Value variables represent a single element of interest to the programmer that takes the form of either a value (a number, date, or string) or an object (a set of key-value pairs). Set variables represent a collection of *value* elements for which the programmer needs to understand the underlying distribution.

To represent objects in both value and set data, we perform an object simplification in which the object is represented by one of its properties (Figure 3). We select a representative property of the object by identifying which of its properties is most commonly used within the code. Visualizations of objects are differentiated from others by color to help avoid misunderstandings about the type of object variables.

Level of Detail: Data

For the *data* level of detail, the programmer is interested in understanding the exact value and underlying distributions of her program variables. We thus identified several visualizations to highlight properties of the data at different temporalities.

Value-Snapshot: Exact Value

When viewing a snapshot of the runtime behavior, we display the exact value `1980` of the variable. For variables representing a single object, we produce a simplified representation that shows a single property of the object `{ "x": 42, ... }`; the full object can be viewed on-demand `{ "x": 42, "y": 56 }`.

Rationale. The programmer often has expectations about the type or value of variables. Displaying the exact value makes it easy to determine whether or not the value is of the expected type or near the expected value. Development environments often enable this check as an on-demand tooltip showing the current value [12]; whereas other development environments require the user to view this information via mouseover, our representation eliminates the need for interaction by surfacing the same information automatically.

Value-Sequence: Line and Horizon

Sequence representations emphasize comparisons across the history (or a subset of the history) of the program runtime. We identified two visualizations to show sequence data: a line chart and a horizon chart [17].

Design. For numeric values, we display the exact value over time. For categorical values, we position each category at its own point on the y-axis, which allows the programmer to view trends in the visitation history; for example, a sawtooth pattern indicates habitual revisitation of an earlier state. For arrays of values, we create a line for each element in the array, based on the value type.

The length of the history can quickly surpass the number of states that a programmer can easily reason over; in response, we found it useful to limit the number of states visualized as the program executes to a subset of the most recent states. We found that visualizing up to twenty states with the horizon chart provides a reasonably interpretable view of the data, using four layers (for both positive and negative values) [17]. However, the programmer can expand the time window on-demand to view a larger slice of the history.

Rationale. The horizon chart is particularly useful for comparisons across positive and negative values [17], whereas the line chart provides a more easily interpretable view of the overall trend. For categorical values, the line chart may be useful for visualizing patterns in the visitation history, but does not otherwise encode useful information in the y position.

Set-Snapshot: Histogram and Heatmap

To provide an overview of set variables, we identified two visualizations of the underlying distribution of set data: a histogram and a heatmap.

Design. For sets of numeric values, we arrange the values into uniformly sized bins. For categorical data, we compute distinct bins for each category and visualize the top n. We place all remaining values in a separate ("other") bin and visualize it alongside the top n. The "other" bin is colored black and is drawn to scale *up to* the size of the largest bin in the top n. This representation allows the programmer to make comparisons among the largest bins while still representing all the data. In the histogram, the set contains 8 different values in varying quantities. We visualize the top n (where $n = 6$) and thus place two values into the "other" bin.

We found that visualizations with a size of about eight pixels per bin are easily interpretable (as in the examples above) and can support interaction on the elements. Representations that push bars towards a width of two pixels make it harder to distinguish between bins or interact effectively.

Rationale. We recommend the histogram as the position encoding is a more effective representation than the color encoding in the heatmap [24]. We include the heatmap for its amenability to miniaturization, which we discuss later in the paper.

Set-Sequence: Line and Horizon

For set variables, the sequence representation needs to aggregate the underlying results to provide an informative summary of the behavior of the set variable over time. Similar to the *value* type, we visualize the aggregated sequence data as either a line chart or horizon chart.

Design. There are multiple ways to represent the value of set variables at the sequence temporality. For this work, we compute the variance of the dataset and visualize the difference in the variance between the current and previous points in the runtime behavior. Numerous aggregation measures could be applied, and the utility of these measures is highly dependent on the programmer's task and requirements for the dataset. As such, the programmer can configure the system to use the appropriate aggregation measure for her task.

Rationale. Our decision to use the *difference* in the variance was selected to show large shifts in the underlying distribution of the data between states in the program runtime. However, if the difference between states is of less interest, standard aggregations (e.g., mean or median) may be more appropriate. For set representations using the difference measure, we recommend using the horizon chart as it more strongly emphasizes large values and the direction of the change [17]. Horizon charts also provide a more easily interpretable view of small differences given the small size of the visualization.

Level of Detail: Change

At the *change* level of detail, the *value* and *set* data types are simplified to an indication of the level of change for the variable. For *value* variables, the *change* is a boolean indicator of whether or not the variable was updated between snapshots of the program runtime. For *set* variables, the *change* is defined as the number of elements that were added, modified, or removed. The *change* level of detail helps to attract the programmer's attention to variables with dynamic updates.

Value-Snapshot: Indicator

The indicator shows whether a variable has changed in a particular snapshot (or period) of the runtime behavior.

Design. The indicator is either empty when no change has occurred or filled when a change has occurred within snapshots of the program runtime. The indicator can be extended to provide additional information by displaying an arrow indicating the direction of the change or other glyphs.

Rationale. The indicator acts as a midpoint between the *snapshot* and *sequence* temporalities by showing a comparison of the current and previous states, or as a summary of the change in a set of states. This augmentation is the simplest proposed and provides a small indication of where the programmer may want to focus her debugging efforts when understanding changes in the runtime behavior.

Value-Sequence: Tick and Timeline

We identified two visualizations to show the history of changes to value variables: tick and timeline .

Design. The timeline visualization shows a bar for each state where the variable is updated. In the tick visualization, a teal block is shown when a variable is updated and a brown block is shown for states when the variable is not updated.

Rationale. The tick visualization is based on Tufte's baseball sparkline [35], whereas the timeline is motivated by the timeline from Hoffswell et al. [18]. For the change level of detail, we recommend the tick visualization as it provides a clearer indication of the behavior for every snapshot in the sequence. The tick visualization allows the programmer to inspect the value of a variable even when it has not been recently updated; for example, the programmer may be interested in the value when it is used but not updated (e.g., the variable represents a previously defined value and is only "read"). The redundant position/color encoding allows the programmer to easily extract the update status at a glance while also facilitating miniaturization. The timeline visualization may be more appropriate when the variable does not have a value during states when it was not updated (e.g., the variable only exists for states when it has been newly defined).

Set-Snapshot: Modification Indicator

Similar to the indicator, the modification indicator shows changes to *set* variables by counting the number of elements added, modified, or removed at the current snapshot.

Design. The modification indicator creates a bar representing the number of *values* that were added , modified , or removed in the data. The starting point in the dataset labels all values as "added," and displays changes from that point.

Rationale. This representation allows the programmer to understand the impact of transformations on *set* data at a more granular level than whether or not a change has occurred.

Set-Sequence: Stacked Area

We selected the stacked area chart to show the changes within a set variable at the sequence temporality.

Design. The stacked area chart creates a band representing the number of *values* that were added , modified , or removed in the data at each snapshot within the sequence.

Rationale. This representation allows the programmer to see information about the impact of transformations on set data over time. In particular, this behavior can help programmers understand when changes to a set variable are particularly expensive due to unnecessary additions or removals.

Miniaturizations

To reduce the **obtrusiveness** of the code augmentations, we designed miniaturizations for the visualizations. The miniaturization for the horizon chart and line chart is a smaller version of the horizon chart that appears as an underline of the text. The miniaturization for both the histogram and heatmap is a smaller version of the heatmap that appears as an underline of the text. For the tick and timeline visualizations, the miniaturization is a compacted version of the full visualization that appears as an underline of the text. We also use the compacted version of the tick visualization as the miniaturization of the stacked area chart. Each of these miniaturizations was designed to provide useful information similar to that of the larger chart that is amenable to the smaller size constraints. As the indicator is already small, and the exact value itself is important, we do not provide additional miniaturizations.

Interaction

We include a number of interactions to facilitate analysis of the visualizations and facilitate **comparisons** across representations. For the exact value representation, mousing over a simplified object augmentation `{ "x": 42, ... }` displays the full object `{ "x": 42, "y": 56 }`. For the line chart and horizon chart, mousing over the visualization shows a cursor and the value at the current point . To facilitate comparison between augmentations, holding shift draws a cursor and value for the current snapshot across all visible charts in gray. For augmentations representing an object simplification, the cursor displays the property name used for the visualization in addition to the full value .

For the histogram and heatmap, hovering over a bar shows additional details, including the range or value for the current bin and the number of elements. For visualizations representing a simplified object, the augmentation similarly shows the bin and count, but also shows the name of the property that is currently visualized (Figure 3). If the programmer holds shift while interacting with the histogram, the environment updates all the related visualizations to show the distribution *relative* to the programmer's current selection (Figure 1).

Mousing over the tick and timeline visualizations highlights the snapshot and displays the value of the variable at that snapshot . To facilitate comparisons, holding shift highlights and displays the value for all visible augmentations. For the modification indicator and stacked area visualization, mousing over the visualization shows the number of elements, added, modified, or removed from the set variable .

Visualization Size

To support the **comparability** of augmentations throughout the code, we standardize the augmentation width, which is set

Placement	Visualization	Reflow	Spacing	Occlusion	Width	Alignment
Right	99 "x": {"scale": "x", "field": "date" , "offset": 2}, 100 "y": {"scale": "y", "field": "indexed_price" },	●	○	○	○	○
Left	99 "x": {"scale": "x", "field": "date", "offset": 100 "y": {"scale": "y", "field": "indexed_price"},	●	○	○	○	○
Above	98 "update": { 99 "x": {"scale": "x", "field": "date", "offset": 2},	○	●	○	○	○
Below	99 "x": {"scale": "x", "field": "date", "offset": 2},	○	●	○	○	○
Inline Transparent	99 "x": {"scale": "x", "field": "date", "offset": 2}, 100 "y": {"scale": "y", "field": ' },	◐	○	●	●	○
Inline Opaque	99 "x": {"scale": "x", "field": ', "offset": 2}, 100 "y": {"scale": "y", "field": "indexed_price"},	◐	○	●	●	○
Expand Inline	99 "x": {"scale": "x", "field": "date", "offset": 2}, 100 "y": {"scale": "y", "field": ' },	○	○	●	●	○
Expand Below	99 "x": {"scale": "x", "field": "date", "offset": 2}, 100 "y": {"scale": "y", "field": 'ed_price"},	○	○	●	●	○
Left Margin	99 "x": {"scale": "x", "field": "date", "offset": 2}, 100 "y": {"scale": "y", "field": "indexed_price"},	○	○	◐	○	●
Right Margin	99 "x": {"scale": "x", "field": "date", "offset": 2}, 100 "y": {"scale": "y", "field": "indexed_price"},	○	○	◐	○	●
Inline Start	99 "x": {"scale": "x", "field": "date", "offset": 2}, 100 "y": {"scale": "y", "field": "indexed_price"},	◐	○	◐	○	◐
Inline End	99 "x": {"scale": "x", "field": "date", "offset": 2} 100 "y": {"scale": "y", "field": "indexed_price"}	○	○	○	○	○

Figure 4. Twelve placement techniques for in situ visualizations shown within a Vega [32] specification. Each technique has different trade-offs regarding the amount of inline text reflow, additional line spacing, occlusion of other text, variable width requirements, and alignment of the augmentations within the development environment. A cursor is shown for placement techniques requiring interaction (hand cursor shown here for clarity; in a real implementation the cursor is hidden mid-interaction to aid chart reading). Properties are labeled as required ●, up to the discretion of the implementor ◐, or not required ○.

to a constant value based on the average token size in the programming language. The augmentation height is determined by the line height. We add additional inter-word and inter-line space as necessary. For certain placement techniques, we use a variable width as the miniaturization and augmentations appear in place. The standard size allows the programmer to more easily compare between *sequence* augmentations as the visualized time scale is the same [30]. Goffin et al. [10] discuss design considerations for increasing the size of the visualizations to fill the inter-line space or to add additional inter-word padding in more detail.

Visualization Color

We vary the color for the code augmentation based on the *type* of data being displayed in order to facilitate identification of the type when visualizations are similar. For each *type* of visualized data, we select a diverging color scheme to encode positive and negative values in the visualizations described in the preceding sections. For augmentations that show object simplifications, we further differentiate the color to attract the programmer's attention to the simplification (e.g., `index_term` in Figure 3). The decision to change the color based on the *type* of the data makes it easier to differentiate between *value* and *set* data, as *value* data often represents exact values whereas *set* data performs some aggregation on the underlying data. Further differentiating the color for object simplifications ensures that they stand out from both *value* and *set* variable augmentations.

Placement of Code Augmentations

We identified twelve techniques for the placement of code augmentations (Figure 4) and assess each technique based on a set of design trade-offs. These placement options are an extension of the techniques presented by Goffin et al. [10].

Placement Techniques

The *right* and *left* placements position the augmentation immediately adjacent to the corresponding token. The *above* and *below* placement techniques ensure that the individual lines of code maintain their original structure, but may introduce new whitespace lines and increase the overall code length. The *inline-transparent* and *inline-opaque* placements draw the augmentation over the corresponding token, thus requiring interaction to improve legibility for the visualization or code.

To satisfy the need for **unobtrusive** augmentations, we include the placement techniques: *expand-inline* and *expand-below*. These placement techniques require the augmentation to include a miniaturization that displays the augmentation as an underline of the text. Hovering over the token expands the augmentation as either *inline-opaque* or *below*. However, unlike *below*, *expand-below* does not add a whitespace line and instead draws the augmentation over the existing code so as not to require additional *spacing* in the code.

The *inline-start* and *inline-end* placement techniques position augmentations on the same line, but not adjacent to their corresponding tokens. The *inline-start* placement leverages the code indentation to place augmentations in the whitespace at the start of the line. The *inline-end* placement maintains the full readability of the line, using the augmentations as the final punctuating marks. As multiple tokens may occur on the same line, the augmentations can either be placed adjacently or overlapping, but some interaction is required to relate augmentations back to their corresponding tokens.

The *left-margin* and *right-margin* placement techniques separate the augmentations from the token by placing them in the margin of the code editor but improve **comparability** across augmentations. When multiple augmentations exist on the same line, they overlap in the margin, thus requiring additional interaction to select the augmentation of interest.

Placement Trade-off Metrics

For each placement technique described in this section, we discuss trade-offs in the application of the technique with respect to the *reflow* requirements, line *spacing*, augmentation *width*, vertical *alignment*, and *occlusion*. We provide a brief description of each metric and include the results in Figure 4.

- *reflow* The code must reflow inline to make space for the augmentation, thus changing the length of the line.
- *spacing* The code must add additional space between lines to include the augmentation, thus changing the length of the document.
- *occlusion* The augmentation partially or fully occludes the code, thus requiring interaction to improve legibility of the augmentation or code.
- *width* The augmentation must be of a variable width to fulfill the placement requirements.
- *alignment* The augmentation will be vertically aligned with other augmentations in the document.

Reflow. The *left* and *right* placement techniques introduce reflow changes inline to make space for the visualization. Depending on the programming language used, these changes

may be minimal due to the amount of existing whitespace and structure in the code. The *inline-transparent* and *inline-opaque* placement techniques may reflow for small tokens to improve the legibility of the augmentation by increasing its *width*. If many augmentations are on the same line, the *inline-start* placement may increase the indentation at the start of the line. All other techniques do not cause reflow changes.

Spacing. Only the *above* and *below* placements add additional spacing to the document. The impact of this additional spacing is highly dependent on the structure of the code; for augmentations where whitespace is already available above or below the line, we do not introduce a new line into the code but instead use the existing whitespace (Figure 4, *Above*).

Occlusion. The *inline-transparent* and *inline-opaque* placements occlude the token, thus requiring interaction to improve the legibility of the augmentation or code. For the *expand-inline* and *expand-below* techniques, the augmentations will not *occlude* the code when miniaturized, but will introduce some occlusion when expanded to their full size. The *left-margin*, *right-margin*, and *inline-start* techniques may introduce occlusion if the augmentations overlap when there are more than one on a given line. For the *inline-end* placement, augmentation can be positioned side-by-side to facilitate comparison. The *above* and *below* placements may need to handle occlusion with other augmentations on the same line, as described in [10]. Neither *right* and *left* occludes the text.

Width. The *right*, *left*, *above*, *below*, *inline-start*, *inline-end*, *left-margin*, *right-margin* techniques can use a standard *width* to facilitate comparisons. For the *inline-transparent*, *inline-opaque*, *expand-inline*, and *expand-below* techniques, the *width* must match the size of the token.

Alignment. The *left-margin* and *right-margin* augmentations will be aligned, thus facilitating comparisons. Using the *inline-start* placement can produce augmentations that are *aligned* based on the indentation depth of their tokens. All other techniques will not be aligned due to the original token position.

General Placement Guidelines

Based on our design considerations, the augmentations must be **comparable**, **unobtrusive**, and **salient**, such that they attract the programmer's attention to interesting trends, without detracting from her ability to perform her current task.

The **comparability** of the augmentations is largely affected by their *alignment* and *width*. When the programmer needs to make fine-grained comparisons between augmentations, the *margin-left* and *margin-right* placements are ideal because they maintain the same temporal axis across augmentations.

The **unobtrusiveness** of the augmentations requires minimal changes to the code structure caused by *reflow* or additional *spacing*; large structural changes can be detrimental to the programmer's ability to review the code [38]. *Occlusion* is also relevant to the programmer's ability to read the code or extract information from the augmentations. When the **unobtrusiveness** of the augmentations is most important, we recommend using the *expand-inline* placement technique; this technique provides an indicator of what variables are changing while limiting changes to the visibility of the code.

The **salience** of the augmentations influences how easily the programmer's attention is attracted to particular augmentations of interest. Whereas the *expand-inline* placement provides some indication of variable changes or distributions, it can easily be overlooked as it appears only as an underline. To better attract the programmer's attention, we recommend using the *right* placement technique to produce a large augmentation near the source of the token. For tokens on the periphery of the programmer's focus, the *inline-opaque* technique may be better so as to reduce *reflow* changes to the document.

EVALUATION: AUGMENTING THE VEGA EDITOR

To evaluate the utility of in situ visualizations for program understanding tasks, we conducted a user study with 18 programmers presented with unfamiliar programs written in the Vega visualization grammar [32]. To perform this evaluation, we identified in situ visualizations from our design space relevant to novice users and appropriate for Vega runtime state. We implemented these embedded visualizations as an extension to the online Vega code editor. In this section, we discuss our use of Vega and selected embedded visualizations, study methods, and experimental results. We have included the instructions, session script, task specifications, task questions, and post-task questionnaire in the supplemental material.

Background on Vega

Vega is a declarative grammar for specifying interactive visualizations. The relative simplicity of Vega was convenient for exploring the design space of in situ visualizations and strategies for the placement and automatic incorporation of embedded visualizations. However, many of Vega's constructs are representative of properties of more general languages.

The programmer produces a Vega *specification* in JSON format that describes the data transformations, interactive behavior, and visual appearance of the output visualization. *Signals* are dynamic variables that capture interaction events on the Vega output and can be used throughout the Vega code to parameterize the behavior. *Datasets* represent collections of data tuples and can be representative of more complex data structures with arbitrary nesting and usage throughout the code. References to particular *data fields* extract a property from each tuple in the underlying datasets to be used as variables throughout the Vega specification. The runtime state can be represented by snapshotting the *signal* values.

The visual appearance of the output is specified via *scales*, *axes*, *legends*, and *marks*. *Scales* are functions that map from data values to visual properties, and can be visualized as *axes* and *legends*. *Marks* can be arbitrarily nested and dynamically initialized at runtime, thus representing complex data flows during program execution. *Marks* can be parameterized by both *signals* and *data*, with the help of *scales* to produce reasonable mappings from *data fields* to visual properties.

Implementation in the Vega Editor

To evaluate the utility of embedded visualizations, we identified several in situ visualizations appropriate for Vega vari-

ables. We implemented the visualizations using D3.js and utilized the Monaco [26] API to extract tokens from the Vega code and insert the visualizations (see Figures 1, 3). Prior to inserting the visualizations, we performed a preprocessing step in which additional white space is inserted at the visualization position. We identified two types of variables in the Vega specification to visualize: *signals* (value variables) and *data fields* (set variables). Our implementation of embedded visualizations for the Vega editor is available at https://github.com/uwdata/code-augmentation.

In Situ Visualization Selection

We followed a simple rule-based process for selecting the type of in situ visualization to display for each variable in the Vega specification. For this implementation, we do not currently allow the visualization type to change once selected, though such automatic updates are an important part of future work.

Tooltip value: 1890 Hovering over a data field or signal name shows the current state of the variable on-demand. For data fields, the tooltip shows summary information about the data set (e.g., the min, max, and mean value for that field). For signals, the tooltip shows the current value.

Indicator For signal variables that are not expected to change (e.g., do not include an "update" clause or react to input events), we selected the indicator to deemphasize the signal definition. The indicator also allows quick value extraction.

Line For signal variables where the type at runtime is a number, we select the line visualization because it represents the range and value of the variable over time, rather than a snapshot of the change. For signal variables that are an array of numbers, we use a different line for each element in the array to fully represent the variable.

Tick For signal variables that are initialized to null or are not a number at runtime, we selected the tick visualization to emphasize when changes occur and to show the behavior of the variable over time. We selected the tick visualization rather than the timeline because the tick visualization explicitly represents each state in the program runtime and may thus be more informative for novice users.

Histogram We selected the histogram for data field variables because it visualizes the current state. While the horizon chart can highlight substantial shifts in the historical values of the variable, this visualization summarizes the variable rather than representing the underlying values. The horizon chart is also more likely to be unfamiliar to novice users and may thus require additional training to interpret. The histograms can highlight changes between states by observing shifts in the distribution. Since position encodings are more interpretable than color encodings [24], we chose not to utilize the heatmap visualization.

For the evaluation, we selected the *right* placement to ensure that the in situ visualizations are salient and in close proximity to their corresponding token. This placement choice was motivated by Zellweger et al.'s [38] discussion of the importance of proximity for embedding contextual information. Furthermore, the *right* placement reduces the overall reflow of the

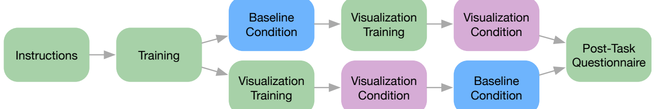

Figure 5. Participants completed two tasks, one in each of two conditions (*baseline* or *visualization*); we counter-balanced the conditions across participants. This figure shows the steps followed in the evaluation, with each participant completing only one path.

document and follows the reading direction of the code. We selected this set of visualizations to provide ones that were likely to be familiar to our novice users and thus interpretable with minimal training. However, other visualization designs may be more useful for specialized tasks by expert users. For example, the modification indicator or stacked area chart could be useful for visualizing tuple-specific changes to the underlying Vega datasets, which is useful for particular debugging tasks surrounding data transformations. As such, we did not evaluate these visualization types.

Participants

We recruited 18 students (7 female, 11 male) from our university, including both PhD (4) and undergraduate (14) students. Participant ages ranged from 18 to 30 ($\mu = 21.1$, $\sigma = 3.74$). Participants completed a screening survey about their experience and programming language familiarity to ensure all participants had prior programming coursework or job experience. The most common programming language regularly used by our participants is Java, followed by Python, JavaScript, and C/C++. All of our participants were novice Vega users (i.e., were unfamiliar with Vega), though two participants had previously seen Vega in other contexts. Each participant received a $20 gift card for completing a 90 minute session.

Methods

Participants answered program understanding questions about two Vega programs, with and without the assistance of in situ visualizations of program behavior.

At the start, participants were given an instruction sheet with a sample Vega program, an explanation of the code, and an introduction to the development environment and important keywords. Participants were then given a training task in which they answered several sample questions and viewed the sample answers. During this time, participants were encouraged to ask any questions about Vega or the task setup. Once participants started the tasks, the researcher no longer answered questions.

For the study tasks, we selected four Vega programs that cover a range of visualization designs, datasets, and program understanding challenges, three of which exhibit an error.

Population. A population pyramid with a slider to select the year that is visualized. This Vega specification includes missing data for the year 1890, which causes derived data sets to be empty and the visualization to be blank at this point.

Index. A line chart of stock prices with an interactive cursor that selects an index point to which stocks are normalized to show investment returns. The Vega specification includes derived datasets and nested data declarations. An error in one

of the data transformations causes all tuples to be filtered out at certain times, causing the lines to visually flatline.

Scatterplot. A scatterplot of points that supports infinite panning. This Vega specification includes many interconnected signal definitions to define the interaction; due to a bug with the evaluation order of signals, the domain of the axes becomes distorted while panning. This example is similar to the one described in the **Usage Scenario**.

Overview. Two area charts showing stock price over time; selecting a region in the smaller chart zooms the larger one. This Vega specification includes many interconnected signal definitions nested in the specification. There is no error.

Participants completed two tasks, one in each of two conditions: *baseline* and *visualization*. In the baseline condition, participants were given a simple code editor based on the online Vega editor. Tooltips were added to the signal and data field tokens in the code to show information about the runtime state, similar to other common development environments. For the *visualization* condition, the editor was additionally augmented with our in situ visualizations of the program behavior. Prior to the *visualization* condition, participants were shown an instruction sheet with an explanation of the in situ visualizations and were encouraged to experiment with them using the same visualization seen in the training task. We counterbalanced the order of the conditions across participants. The experimental protocol is shown in Figure 5.

Participants answered 18 program understanding questions for each task about major Vega concepts, such as *signals*, *data sets*, and *data fields*, which required participants to reason about how the state of the visualization changed during interaction. These questions encouraged participants to read and experiment with the program to develop an understanding of the interconnectedness and runtime behavior of the code. Participants were also asked to identify any unexpected behavior in the Vega output and answer follow-up questions about the source of that unexpected behavior; participants were not informed that an error existed if they did not identify it themselves. Participants provided free-form answers for each question and a rating of their confidence on a scale from 1 (not confident) to 5 (extremely confident).

Participants finally completed a post-task questionnaire in which they rated their self-perceived speed and accuracy on the task questions on a scale from 1 (better with baseline condition) to 7 (better with visualization condition). Participants also scored how helpful, interpretable, and intrusive each of the in situ visualizations were on a scale from 1 (not) to 5 (extremely). The larger scale for the speed and accuracy was selected to encourage nuanced comparison of the conditions.

Quantitative Results & Analysis

To perform the analysis, we first created a gold-standard set of answers for each task and scored participant answers on an integer scale from 0 to 2 (“incorrect,” “partially correct,” “correct”). Scores were provided by the second author, who was blinded to the study condition for each task. Final grades were determined by simple summation.

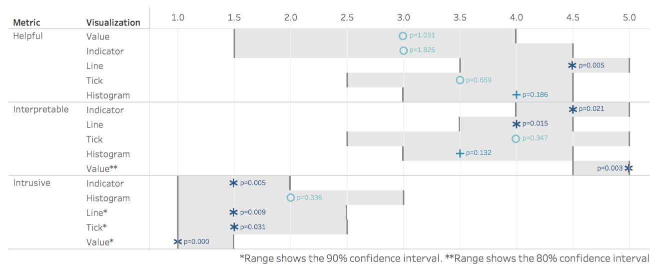

Figure 6. The pseudo-median value and 95% confidence interval (unless otherwise noted) for how helpful, interpretable, and intrusive each in situ visualization was on a scale from 1 (not) to 5 (extremely). Median values are labeled with the p-value for the 1-sample Wilcoxon signed rank test. Note: The Wilcoxon rank test could not compute the full 95% confidence interval for scores tightly clustered near one or five. We include the R script for computing the statistics in supplemental material.

We fit linear mixed-effects models for participants' grades, log-transformed task times, and average confidence. Each model included fixed effects for condition and presentation order, plus per-subject random intercepts. Likelihood ratio tests indicated a marginally significant effect of the visualization condition on task grade ($\chi^2(1) = 3.30$, $p < 0.1$). Participants had roughly one more “correct” (or two more “partially correct”) answers in the visualization condition overall. Exploratory data analysis also indicated a strong difference in grades due to education level, but with similar absolute grade improvements in the visualization condition. There was a significant effect of task order on the log time for participants to complete the task ($\chi^2(1) = 8.96$, $p < 0.01$), with participants faster in the second task regardless of condition. We found no significant effect of condition or order on participant confidence.

For the post-task questionnaire, we used 1-sample non-parametric Wilcoxon signed rank tests with a null hypothesis that the result is neutral (middle Likert scale value). We found significant positive effects in favor of the in situ visualizations for the participants' self-reported speed ($p = 0.002$) and accuracy ($p = 0.0026$).

Figure 6 depicts subject ratings of how helpful, interpretable, and intrusive each of the visualizations were. We found a significant positive effect for line visualization helpfulness ($p = 0.005$) and a marginally positive effect for histogram helpfulness ($p = 0.186$). We found a significant positive effect for how interpretable the value ($p = 0.003$), indicator ($p = 0.021$), and line ($p = 0.015$) visualizations were, and a marginally significant positive effect for the histogram ($p = 0.132$). For the intrusiveness of each visualization, we found a significant negative effect for the value ($p = 0$), indicator ($p = 0.005$), line ($p = 0.009$), and tick ($p = 0.031$).

Qualitative Results & Discussion

We selected questions that would be reasonable to expect participants to answer regardless of condition. The notable significant effect of task order on the completion time suggests some improved knowledge of the language and questions, which helped participants know where to look. While we did see a marginally significant effect of the visualization condition on task grade, participants were generally able to answer the task questions by reviewing the code and probing the state information with the tooltip on signals and data fields.

The question answering process could be quite different between the two conditions. Question 8 asked participants to identify how each signal that updates is used throughout the code. In the baseline condition, P18 spent over 16 minutes attempting to identify how each signal in the specification behaved (Q8 median 3.65 min). In order to fully answer this question, P18 carefully experimented with different interactions, probing the signal value with the tooltip to identify when in changed. This back and forth between testing interactions with the output and assessing the state clearly demonstrated the disconnect between the code behavior and output.

Participants in the visualization condition similarly tested interactions, but could identify changes at a glance. As P2 put it: *"the [in situ] visualizations allowed me to connect the dots between the code, its properties, and what it did."* When comparing the two conditions, P11 noted that *"The biggest factor for me was just seeing which values change in real-time when interacting with the visualization."* Rather than reading the code or probing the state on-demand, participants could view changes as they worked rather than as a separate task.

We saw a significant positive effect in favor of the in situ visualizations on both participants' self-reported speed and accuracy. Across conditions, participants utilized search to find keywords of interest. But, as P15 explains, *"the code visualizations helped better locate the signals and made me more confident about my answers."* Moreover, the in situ visualizations turned the underlying data into a physical artifact to reason over. P2 noted that *"I found it helpful to be able to interact with the data on a graphical, physical level."*

The error in the index chart can be a challenging one to reason about because it occurs due to a small difference in the date/time of the tuples in the dataset from the filter window. For most participants, it was clear that the error only occurred at certain dates in the visualization. However, the error was not with the signal itself. For P9, it became clear that the error was in the data because *"when I mouse over the flatline behavior, the 'index' field changes and the [histogram] shifts to the left (next to the variable)."* When interacting with the in situ visualization of the signal, P9 also correctly noticed that that indexDate contained a time in addition to the date. While none of the participants correctly diagnosed the error in its entirety, P9's observations were crucial steps towards uncovering the convoluted source of this error. The in situ visualizations provide a lightweight way to incorporate the underlying state into the program understanding process.

While the utility of in situ visualizations for interpreting interactive behavior was apparent, participants sometimes struggled to understand their contextual implications, particularly for the data fields. For instance, to understand the range of different data fields in the index chart, P9 inspected the visual encodings and noted *"Oh, okay, I guess Microsoft... for some reason."* Although P9 identified that the data field only had one value (which was Microsoft), it was not clear why this was the case when other parts of the code showed the full range (e.g., all five companies). In this case, P9 had overlooked the nested dataset declaration; five different marks are dynamically created, but the embedded visualizations only show the results for one of them. To understand where each data field came from, participants needed a more intimate understanding of the implementation hierarchy and dataflow.

LIMITATIONS AND FUTURE WORK

While Vega's relative simplicity was convenient for evaluating the design space of in situ visualizations, we believe that Vega is an exemplar of a larger class of reactive languages, such as React [19] and Elm [7], for which these techniques could apply. For example, Elm [7] was originally designed as a reactive programming language that similarly utilizes the streaming constructs implemented (and visualized) in Vega.

Moreover, many of Vega's constructs are representative of properties seen in other languages. The arbitrary nesting of mark definitions produces numerous examples of how scope can be a concern when referencing particular variables and how such environments are dynamically allocated at runtime. Since marks can be dynamically added and removed, the scope and number of instances can change as the programmer interacts. While the design space presented in this paper provides a breakdown of potential data types of interest, we do not currently address this underlying code structure. Furthermore, the use of object simplification and summarization in the embedded visualization shows one approach to handling complex nested data structures by emphasizing particular properties of interest. Future work should explore how interaction or other techniques might surface relevant scoping information or navigate complex nested data structures.

Vega's reactive framework was useful for effectively snapshotting the program behavior for producing the in situ visualizations. While we do not propose advances in system logging, techniques such as Dolos from Burg et al. [5] enable efficient logging with minimal overhead for web programming. This infrastructure could provide an effective framework on which to introduce in situ visualizations for web development.

While in situ visualizations helpfully call attention to the dynamic behavior of program variables, it may be important to change their salience relative to the programmer's current task. P14 noted that *"I thought the code visualizations were cool and helpful, but they could also be a little distracting."* Future work should consider techniques to infer the programmer's task and deploy in situ visualizations as needed.

In this paper we described the design space for embedded visualizations of program behavior, including design considerations for the placement of such visualizations within code. In an evaluation with novice Vega users, we found that participants improved their overall task grade by about 2 points when using the in situ visualizations and exhibit significant positive effects on their self-reported speed and accuracy.

ACKNOWLEDGMENTS

We thank the reviewers, the UW Interactive Data Lab, and many others for their helpful comments. This work was supported by the National Science Foundation (IIS-1758030) and Moore Foundation Data-Driven Discovery Investigator Award.

REFERENCES

1. Fabian Beck, Fabrice Hollerich, Stephan Diehl, and Daniel Weiskopf. 2013a. Visual monitoring of numeric variables embedded in source code. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*. IEEE, 1–4.
2. Fabian Beck, Oliver Moseler, Stephan Diehl, and Günter Daniel Rey. 2013b. In situ understanding of performance bottlenecks through visually augmented code. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 63–72.
3. Richard A Becker and William S Cleveland. 1987. Brushing scatterplots. *Technometrics* 29, 2 (1987), 127–142.
4. Mike Bostock. 2014. Visualizing Algorithms. `https://bost.ocks.org/mike/algorithms/`. (2014).
5. Brian Burg, Richard Bailey, Andrew J Ko, and Michael D Ernst. 2013. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. ACM, 473–484.
6. Bay-Wei Chang, Jock D Mackinlay, Polle T Zellweger, and Takeo Igarashi. 1998. A negotiation architecture for fluid documents. In *Proceedings of the 11th annual ACM symposium on User interface software and technology*. ACM, 123–132.
7. Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. In *Proc. ACM SIGPLAN*. ACM, 411–422.
8. Camil Demetrescu, Irene Finocchi, and John T Stasko. 2002. Specifying Algorithm Visualizations: Interesting Events or State Mapping? In *Software Visualization*. Springer, 16–30.
9. Pascal Goffin, Jeremy Boy, Wesley Willett, and Petra Isenberg. 2016. An Exploratory Study of Word-Scale Graphics in Data-Rich Text Documents. *IEEE Transactions on Visualization and Computer Graphics* 99 (2016), 1.
10. Pascal Goffin, Wesley Willett, Jean-Daniel Fekete, and Petra Isenberg. 2014. Exploring the placement and design of word-scale visualizations. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2291–2300.
11. Pascal Goffin, Wesley Willett, Jean-Daniel Fekete, and Petra Isenberg. 2015. Design considerations for enhancing word-scale visualizations with interaction. In *Posters of the Conference on Information Visualization (InfoVis)*.
12. Google. 2018. JavaScript Debugging Reference. `https://developers.google.com/web/tools/chrome-devtools/javascript/reference`. (January 2018).
13. Scott Grissom, Myles F McNally, and Tom Naps. 2003. Algorithm visualization in CS education: comparing levels of student engagement. In *Proceedings of the 2003 ACM symposium on Software visualization*. ACM, 87–94.
14. Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for CS Education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 579–584. DOI: `http://dx.doi.org/10.1145/2445196.2445368`
15. Matthew Harward, Warwick Irwin, and Neville Churcher. 2010. In situ software visualisation. In *Software Engineering Conference (ASWEC), 2010 21st Australian*. IEEE, 171–180.
16. Christopher Healey and James Enns. 2012. Attention and visual memory in visualization and computer graphics. *IEEE transactions on visualization and computer graphics* 18, 7 (2012), 1170–1188.
17. Jeffrey Heer, Nicholas Kong, and Maneesh Agrawala. 2009. Sizing the horizon: the effects of chart size and layering on the graphical perception of time series visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1303–1312.
18. Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2016. Visual Debugging Techniques for Reactive Data Visualization. *Computer Graphics Forum (Proc. EuroVis)* (2016). `http://idl.cs.washington.edu/papers/vega-debugging`
19. Facebook Inc. 2017. React: A JavaScript Library for Building User Interfaces. `https://reactjs.org/`. (2017).
20. Andrew J Ko and Brad A Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 151–158.
21. Andrew J Ko and Brad A Myers. 2005. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing* 16, 1 (2005), 41–84.
22. Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering* 32, 12 (2006).
23. Tom Lieber, Joel R Brandt, and Rob C Miller. 2014. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*. ACM, 2481–2490.
24. Jock Mackinlay. 1986. Automating the design of graphical presentations of relational information. *Acm Transactions On Graphics (Tog)* 5, 2 (1986), 110–141.
25. D Scott McCrickard and Christa M Chewar. 2003. Attuning notification design to user goals and attention costs. *Commun. ACM* 46, 3 (2003), 67–72.

26. Microsoft. 2017. Monaco Editor. https://microsoft.github.io/monaco-editor/index.html. (2017).

27. Jeanne Nakamura and Mihaly Csikszentmihalyi. 2014. The concept of flow. In *Flow and the foundations of positive psychology*. Springer, 239–263.

28. Stephen Oney and Brad Myers. 2009. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 105–108.

29. Chris Parnin and Spencer Rugaber. 2011. Resumption strategies for interrupted programming tasks. *Software Quality Journal* 19, 1 (2011), 5–34.

30. Zening Qu and Jessica Hullman. 2018. Keeping Multiple Views Consistent: Constraints, Validations, and Exceptions in Visualization Authoring. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2018). http://idl.cs.washington.edu/papers/consistency

31. David Saff and Michael D Ernst. 2003. Reducing wasted development time via continuous testing. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. IEEE, 281–292.

32. Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2015. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2015). http://idl.cs.washington.edu/papers/reactive-vega-architecture

33. Clifford A Shaffer, Matthew Cooper, and Stephen H Edwards. 2007. Algorithm visualization: a report on the state of the field. In *ACM SIGCSE Bulletin*, Vol. 39. ACM, 150–154.

34. Ben Swift, Andrew Sorensen, Henry Gardner, and John Hosking. 2013. Visual code annotations for cyberphysical programming. In *Proceedings of the 1st International Workshop on Live Programming*. IEEE Press, 27–30.

35. Edward R Tufte. 2006. Beautiful evidence. *New York* (2006).

36. Bret Victor. 2012. Learnable programming: Designing a programming system for understanding programs. http://worrydream.com/LearnableProgramming. (2012).

37. Wesley Willett, Jeffrey Heer, and Maneesh Agrawala. 2007. Scented widgets: Improving navigation cues with embedded visualizations. *IEEE Transactions on Visualization and Computer Graphics* 13, 6 (2007), 1129–1136.

38. Polle T Zellweger, Susan Harkness Regli, Jock D Mackinlay, and Bay-Wei Chang. 2000. The impact of fluid documents on reading and browsing: An observational study. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, 249–256.