Live, Synchronized, and Mental Map Preserving Visualization for Data Structure Programming

Akio Oka
Tokyo Institute of Technology
School of Computing
Tokyo, Japan
a.oka@prg.is.titech.ac.jp

Hidehiko Masuhara
Tokyo Institute of Technology
School of Computing
Tokyo, Japan
masuhara@acm.org

Tomoyuki Aotani
Tokyo Institute of Technology
School of Computing
Tokyo, Japan
aotani@is.titech.ac.jp

Abstract

Live programming is an activity in which the programmer edits code while observing the result of the program. It has been exercised mainly for pedagogical and artistic purposes, where outputs of a program are not straightforwardly imagined. While most live programming environments so far target programs that explicitly generate visual or acoustic outputs, we believe that live programming is also useful for *data structure programming*, where the programmer often has a hard time to grasp a behavior of programs. However, it is not clear what features a live programming environment should provide for such kind of programs. In this paper, we present a design of live programming environment for data structure programming, identify the problems of synchronization and mental map preservation, and propose solutions based on a calling-context sensitive identification technique. We implemented a live programming environment called Kanon, and tested with 13 programmers.

CCS Concepts • **Software and its engineering → Integrated and visual development environments**;

Keywords Live programming, data structures, object graph

ACM Reference Format:
Akio Oka, Hidehiko Masuhara, and Tomoyuki Aotani. 2018. Live, Synchronized, and Mental Map Preserving Visualization for Data Structure Programming. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '18), November 7–8, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3276954.3276962

Onward! '18, November 7–8, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6031-9/18/11…$15.00
https://doi.org/10.1145/3276954.3276962

1 Introduction

Live programming has been researched [Hancock, 2003; Victor, 2012] as one way to enable easier programming. Traditional programming is divided into a phase for editing a program and a phase for confirming whether the program is working as expected. The programmer needs to return to editing the program if the execution result of the edited program differs from the expected result.

On the other hand, live programming assists the programmer by giving an "immediate connection" between a program and its execution result without requiring the programmer to run the program in their mind. Most past demonstrations of live programming target programs whose results are not obvious from their texts, including the programs for drawing pictures [Victor, 2012], for synthesizing music [Aaron and Blackwell, 2013], for animating game characters [McDirmid, 2007], and for teaching algorithms [Khan Academy, 2018].

Data structure programs fall into the same category, and therefore we believe live programming can be helpful in this domain as well. By data structure programs, we here mean definitions of data structures and their operations at various levels of abstractions, ranging from generic ones like a doubly-linked list to application-specific ones like "data for a hospital medical record system." In object-oriented programming languages, data structure programs are usually defined as class and method definitions.

We propose a live data structure programming environment that provides the immediate connection between the program text and graphical images of data structures in the programmer's mind[1] . Though the programmers could have a variety of mental images for data structures, we assume that images with boxes and arrows are common enough. Figure 1 is an example of such an image for a doubly-linked list.

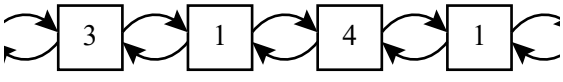


Figure 1. A mental image of a doubly-linked list.

[1]The early stages of the work is reported as the workshop papers [Oka et al., 2017a,b,c].

While it is a straightforward idea and there is a tremendous amount of research that visualizes data structures, it is not obvious what features programming environments should provide in the context of live programming. Though there are many programming environments, like ZStep [Lieberman and Fry, 1995], jGRASP [Hendrix et al., 2004] and Python Tutor [Guo, 2013], that visualize user-defined data structures, they mainly focus on the situation when the developer tries to examine the behavior of programs in a *post-mortem* fashion. In other words, development and examination are separated processes in those environments.

In order to provide a more live experience of data structure programming, we need to consider the problem of visualization while the developer is writing and changing a fragment of code[2]. In the following sections, we discuss the background of this research (Section 2). We then describe the design of our proposed programming environment, Kanon (Section 3) and its implementation issues (Section 4). We report a summary of our findings when we let the developers use Kanon (Section 5, an excerpt of Appendix A). Finally, we discuss related work (Section 6) and conclude the paper (Section 7).

2 Background

2.1 Data Structure Programming

Data structure programming is the act of programming data structures as well as operations that manipulate those structures. Data structures include common ones like lists and problem specific ones, and appear in many programs.

Data structure programming is sometimes difficult and frustrating, as it is often involved with multiple references. When we define an operation on a data structure, the operation needs to take several steps to modify the structure such as by changing references. In such a case, we need to think about the next step by imagining the shape of the structure modified by the steps written so far. The problem can be even harder when there is aliasing of references and cyclic references.

When we are defining a complicated operation that manipulates a data structure, we sometimes write a test case and examine the (partly) modified structure. However, textual printouts of data structures, which would be the most widely taken approach, are often hard to read, especially when the structures become complicated. It is also difficult to recognize changes in a data structure from its textual outputs.

[2]While we primarily focus on the activities of writing and modifying code for data structures, we do not exclude the situations of debugging and code-understanding. When writing a new code fragment often involves identifying problems in the written code, and understanding the existing code related to the one being written. This would be especially true with live programming.

2.2 Live Programming Environments

We can classify live programming environments into two groups with respect to the types of the programs they support. The first group's environments enable code editing of a running program (e.g., SonicPi [Aaron and Blackwell, 2013]). The second group's environments automatically re-execute a program to show the effects of changes immediately (e.g., Live Editor [Resig, 2012] and YinYang [McDirmid, 2013]). The former group is mainly used for artistic performances, like improvising music and animated graphics. The latter is mainly used for software development and pedagogical purposes.

We can also classify live programming by the types of outputs from programs. Many environments mainly target programs that generate *visual or acoustic outputs*. Additionally, environments for artistic performances, demonstrations for educational usages often use programs that draw pictures.

A few exercises are reported to use live programming for programs that do not output visual or acoustic outputs. Live Editor [Resig, 2012] and YinYang [McDirmid, 2013], for example, live-update *textual outputs* from a program being edited. These tools are used to show the course of computation taken place in a loop of a numerical function, such as the square root of numbers.

If we apply live programming to data structures programs, the current environments are not suitable for the following reasons: in the first case, the programmer has to write a program that explicitly generates visual, acoustic or textual outputs. This is clearly tedious for operations that manipulate data structures. In the latter, the only standard way to output data structures is printing in text, which is not friendly to the programmer's eyes as we discussed in the previous section.

2.3 Algorithm Animation

Many algorithm animation systems, including Balsa [Brown and Sedgewick, 1984], Zeus [Brown, 1991] and Tango [Stasko, 1989], can graphically display data structures. Some of these systems provide frameworks where we can easily develop an animation by instrumenting an implementation of an algorithm like sorting.

While these algorithm animation systems could be used for program understanding, they are fundamentally different from live programming, as they do not have a *live updating* feature. In other words, they are designed for visualizing behaviors of completed programs; they would not work well for partly-written and frequently edited programs. Though it would be hypothetically possible to automatically apply an algorithm animation system to a program being edited, it would not provide continuous feedback as we will discuss in the later section.

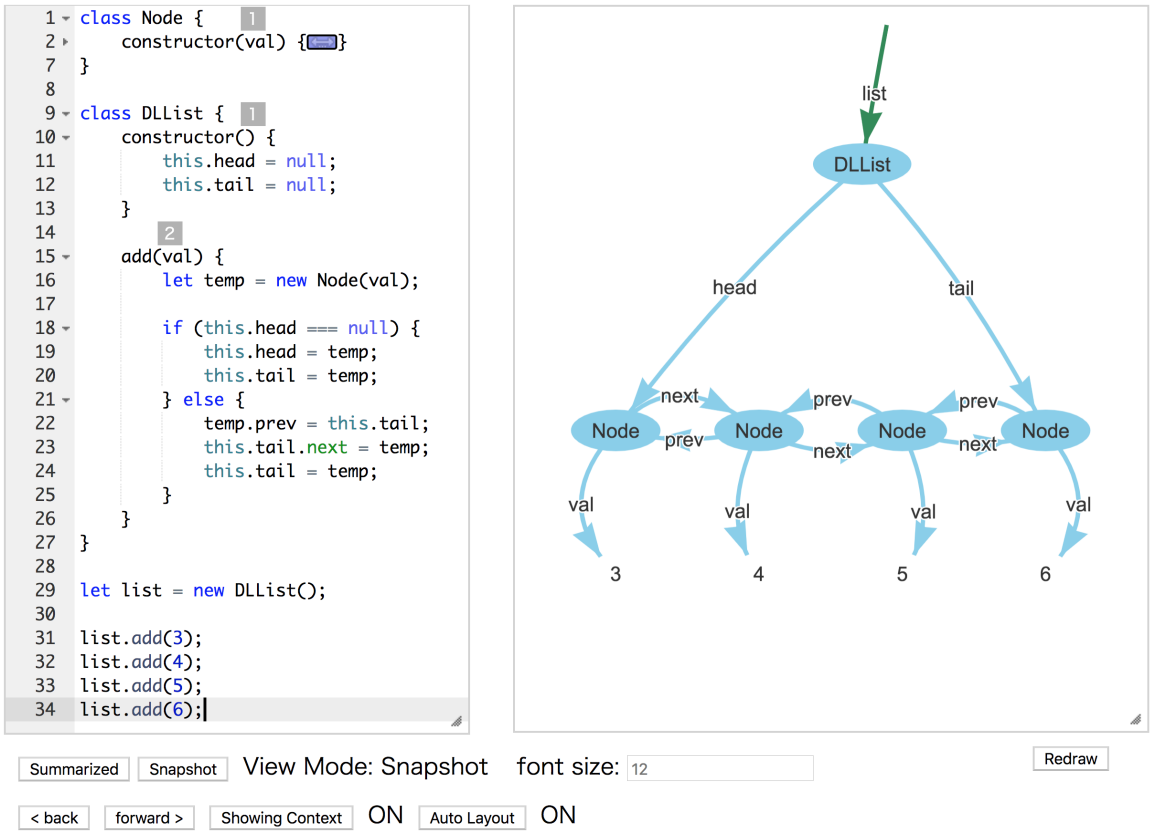


Figure 2. A Screenshot of Kanon.

3 Kanon: A Live Data Structure Programming Environment

We propose a live programming environment, called Kanon, specialized for data structure programming. As the design space for such an environment is very large, we will first discuss our design decisions. We will then introduce two unique features of Kanon.

3.1 Design Overview and Assumptions

Figure 2 is a screenshot of Kanon. The left- and right-hand sides are the editor pane in which a program is written, and a visualization pane that displays data structures, respectively. It is designed under the following assumptions.

- We assume that a program is written in JavaScript[3] in a single file. It consists of definitions of data structures and their operations, followed by top-level expressions that serve as test cases.
- We draw data structures as a node-link diagram. Each oval in the visualization pane represents an object that is created during an execution, labeled with the class name of the object. The blue arrows from the ovals show the field values in the object, which point to either other objects or primitive values, and the green arrows with no origin (e.g. the arrow labeled `list`) show which object the local variables refer to.
- We visualize all objects created from the beginning up to a certain point of execution in a run of a program. Selection of the execution point will be discussed in Section 4.2.

[3]We chose JavaScript as a general-purpose programming language that supports data structures. Therefore, we do not consider use-cases specific to JavaScript, such as DOM and async.

3.2 Visualization of Changes and Two View Modes

3.2.1 Visualization of Changes

When a live programming environment visualizes data, the data can change during execution. For example, given a program that repeatedly approximates a mathematical function, we might want to see the changes of intermediate results during a run. Existing environments can show such changes as a series of values [Imai et al., 2015; McDirmid, 2007] or as a line chart [Apple Computer, 2016]. For programs that produce visual images (i.e., drawing programs), there have been attempts to use a stroboscopic visualization [Granger, 2012] or a timeline visualization [Kato et al., 2012].

Listing 1. Partially defined add.

```
class DLList {...
  // add the given val at the end of the list
  add(val) {
    var temp = new Node(val);
    if (this.head === null) { // when the list is empty
      this.head = temp;
      this.tail = temp;
    } else { // when the list is not empty
      ▮
    }
  }
}
var l = new DLList();
l.add(3); l.add(4); l.add(5);
```

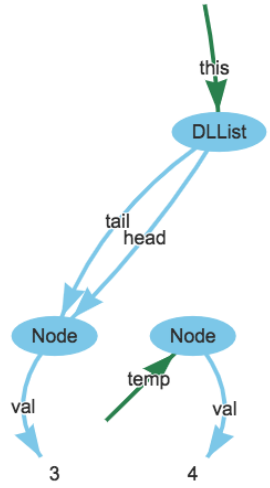


Figure 3. A snapshot view in the context of `l.add(4)`.

Listing 2. Finished (yet incorrect) definition of add.

```
class DLList {...
  // add the given val at the end of the list
  add(val) {
    var temp = new Node(val);
    if (this.head === null) { // when the list is empty
      this.head = temp;
      this.tail = temp;
    } else { // when the list is not empty
      temp.prev = this.tail;
      this.head.next = temp;▮
      this.tail      = temp;
    }
  }
}
var l = new DLList();
l.add(3); l.add(4); l.add(5);
```

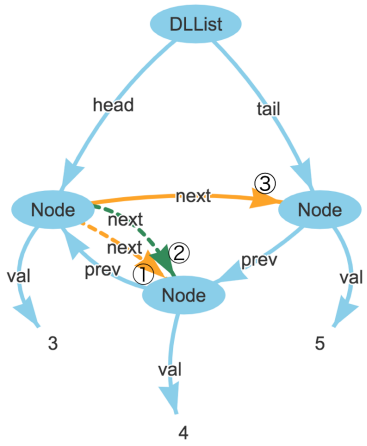


Figure 4. A summarized view for the program in Listing 2.

For data structures, there is no definitive way to visualize changes. Though there have been a number of studies on algorithm animation, those studies tend to develop techniques specialized to specific algorithms.

3.2.2 Snapshot and Summarized View Modes

We provide two ways of showing changes in a program run: one is called the *snapshot view mode*, which animates the graphical representation using cursor movement in the text editor, and the other is called the *summarized view mode*, which shows summarized effects of changes in one graphical representation.

With the *snapshot view mode*, the view shows the object graph with variable references when the program execution reaches the cursor position. If the execution reaches the cursor position multiple times (due to multiple function calls or loops), the execution of a specific *context* is chosen[4].

Figure 3 is an example of a snapshot view for the program text in Listing 1 (in which the cursor position is denoted by a black rectangle), in the context of `l.add(4)`. The green arrows in Figure 3, which are labeled `this` and `temp`, represent the references by the `this` expression and the variables available in the specified calling-context. In this example, the programmer is defining the `add` method for doubly-linked lists and has finished defining the case when the list is empty. The view shows the object graph when the execution of

[4]The current implementation shows just the count of the context specified by each function and each loop. Showing more intuitively is left for future work. Drawing a segmented line that connects the call sites as done in Dr. Racket (https://racket-lang.org/) or the macro visualization in Seymour [Kasibatla and Warth, 2018] are possibilities.

`l.add(4)` reaches the cursor position. Note that the node for 5 is not yet created in this view.

With the *summarized view mode*, the view shows effects of a statement[5] at the cursor position over the object graph at the end of the execution. The view summarizes the effects by two means: (1) when the statement is executed more than once, it visualizes all the effects performed in those executions; and (2) when the statement contains a function call, it visualizes all the effects performed in the call.

Figure 4 is an example of a summarized view for the program text in Listing 2, where the programmer has finished definition of `add`. This view shows an object graph at the end of execution. At the same time, the view illustrates the effects of the code at the cursor position, in this case, the assignment "`this.head.next = temp;`", where the orange solid arrow (③) shows the reference found at the end of execution. The dashed arrows (①, ②) denote the overwritten references, i.e., once created by this (orange ①) or other (green ②) assignment, and then disappeared due to later assignments.

From the diagram, the programmer can observe that the final graph is incorrect. The `next` field of the leftmost `Node` should reference the middle `Node`, where instead it references the rightmost `Node` in the final state. The programmer can also see that the reference was initially correct (as shown with the green dashed arrow ②) and then overwritten by the assignment at the cursor position. In fact, the cursor line should assign to `this.tail.next`, instead of `this.head.next`.

3.3 Backward Connection to Code from Graphical View

3.3.1 Relating Visualized and Program Elements

It is important to the programmer to be able to establish a connection between visualized information and program elements. For example, consider an environment that visualizes a time-series of values of multiple variables in a program. We then need to find out the correspondence between a series of values to a variable, as well as a value in a series to a specific moment in an execution.

YinYang's solution to this issue is a *probe* that displays a value of an expression just below the expression, and a *tracing* construct that produces a clickable output, which rewinds the program state to the time when the output is produced.

For data structures, since a visual representation (i.e., a node-link diagram) has a structure, environments should help to establish a connection between those visual elements and program elements.

[5]Though our current implementation only shows effects of one statement, it is not difficult to extend it to show the effects of a series of statements. It is a part of our future work.

3.3.2 Jump to Construction

We provide a mechanism that helps to connect visual elements to program elements. The mechanism is called *jump-to-construction*, which is invoked by a double-click on a graphical element, and moves the cursor position to the program element that corresponds to the graphical element (either a new expression or a field-assignment statement).

4 Implementation

We implemented a prototype of Kanon for JavaScript running on web browsers. It is available online[6]. Below, we first overview the implementation and then describe a technique to preserve mental map.

4.1 Overview

Figure 5 overviews the implementation. We will explain the structure by following the operations taken place upon a program modification.

We use a modified version of the Ace editor [Jakobs, 2018] for editing a program text. When the programmer edits a piece of text, it notifies the visualization engine.

① The visualization engine uses Esprima [Hidayat, 2018] to parse the program text in the editor. It then traverses the syntax tree by applying the following modifications:

- It inserts declarations of global variables for keeping track of calling contexts and the virtual timestamp.
- For each `new` expression, it appends a piece of code that records object ID in a special field of the created object.
- For each statement and `new` expression, it inserts checkpointing code before and after the statement. The checkpointing code is an expression that applies a list of global and local variables to the object traversal function.
- At the beginning of each loop body and function body, it inserts counting code.

② The engine then evaluates it using `eval`. When the checkpointing code runs, it collects JavaScript objects that are reachable from the variables in the scope. We use the object reflection mechanism to obtain field values from an object. The objects and their references are recorded as graph data (i.e., nodes and links) with a virtual timestamp that increases every checkpointing execution.

③ To update graphical representation, the engine first obtains the cursor position from the editor and then identifies the nearest checkpoint to the cursor position. It then calculates a range of virtual timestamps which corresponds to the current visualization context. Finally, it selects the object graph that is recorded at the nearest checkpoint within the calculated timestamp range.

[6]https://github.com/prg-titech/Kanon (source code), https://prg-titech.github.io/Kanon/ (executable in web browsers)

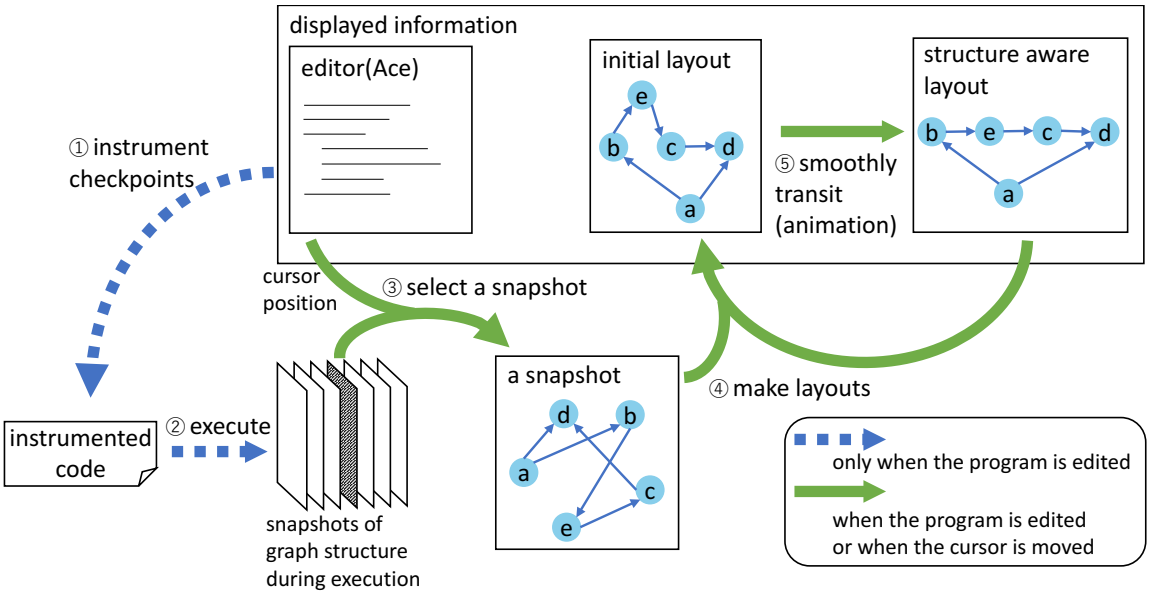


Figure 5. Overview of the implementation.

④ The engine computes the initial layout of the nodes of the objects in the object graph, and draws the graph. The layout is computed by using the currently shown layout (for an older object graph) in combination with a physics-based graph layout algorithm. First, it determines the set of the nodes in the new object graph that are included in the old object graph (explained in Section 4.3). It pins those nodes down to the same geometric locations as in the layout currently shown. Second, it runs a physics-based graph layout algorithm so that the newly created nodes will be placed aesthetically-pleasing positions. We use the vis.js visualization library [B.V., 2018] both for calculating the layout as well as for drawing the resulted layout.

⑤ Finally, the engine computes the structure-aware layout and smoothly moves the visualized graph from the initial layout to the new one. The current algorithm simply recognizes a list or binary-tree structure based on field names, and then places the nodes of the structure on a horizontal line or a tree shape.

4.2 Synchronizing Visualization Context with Cursor

Kanon equips two view modes for visualization of data structures which are changed as execution progresses. Here we explain which object structure is selected in these two view modes as a graph displayed in Kanon.

In the *snapshot view mode*, Kanon displays an object structure stored at the closest checkpoint before the current cursor position. Because the selected checkpoint might be executed multiple times, the user can specify a context of the closest loop or method surrounding the cursor position. In the case of Figure 6, "the closest checkpoint before the cursor position" indicates checkpoint ① and Kanon selects $①_n$ from the specified loop count n.

In the case of the *summarized view mode*, Kanon displays an object structure stored at the final checkpoint. Additionally, Kanon calculates the difference between the object structures stored directly before and after the cursor position for each loop iteration. When the two checkpoints are directly within the same loop or method, we highlight the difference in the graph. In the case of Figure 6, we display the object graph stored at the final checkpoint, namely ③, and highlight the nodes and links that are different either between $①_1$ and $②_1$, and between $①_2$ and $②_2$.

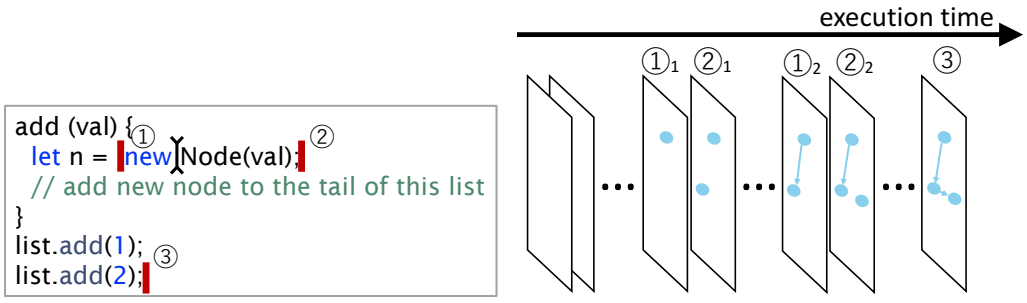
4.3 Mental Map Preservation

4.3.1 Motivation

Live programming environments should *preserve the mental map* when a program is modified. Here, the mental map[7] means a representation in the developer's mind who saw a visual image of a program output. Preservation of the mental map is achieved, when the system displays a visual image of an output of a new program, by keeping the differences of those visual images as small as possible.

For example, adjusting constant parameters in a drawing program is one of the well-known demonstrations of live programming. By immediately executing (i.e., drawing pictures)

[7]The concept of the mental map preservation was proposed for drawing algorithms for dynamically changing graphs [Archambault and Purchase, 2012; Lee et al., 2006]. It should not be confused with the concept of *navigability*, which concerns about the connection between a code fragment and a visualization element in live programming environment [Burckhardt et al., 2013]. We discuss the features related to navigability in Section 3.3.



(a) An example program inserted checkpoints. **(b)** Object graphs stored at each checkpoint.

Figure 6. Which checkpoint is chosen?

a modified program, the programmer can observe the effect of changes as animation. Some environments provide special mechanisms like slider bars for continuously modifying constant values [Khan Academy, 2018].

In Kanon, the mental map preservation means keeping the differences of the graph layouts as small as possible, when it draws a modified object graph. This property is known to be important in the studies of dynamic graph drawing [Archambault and Purchase, 2012; Lee et al., 2006] because the human who saw a graph would need a lot of time to grasp the structure of the graph.

4.3.2 Problem of a Naïve Implementation

It is not a trivial task for Kanon to preserve of the mental map. Assume that a programmer is writing a function `make` that creates a binary tree. Figure 7(a) shows an incomplete function definition that merely creates right children of the tree, which effectively creates a linked-list. The programmer moves the cursor at line 9 in order to insert a piece of code, and sets the visualization context to the second call to `make`, where Figure 7(b) is the object graph at this moment. Note that *the programmer is thinking about a* `Node` *object referenced by a* `next` *field of the root node, whose visual representation is at just right of the root node.*

Now the programmer inserts a statement "`node.left = make(n-1);`" to line 9, which lets the program create a binary tree. The questions are: Where should the nodes of the binary tree be placed? Where should the visualization context be set?

If there were a naïve algorithm that places the nodes created by the modified program based on the order of object creation, its visualization would be like Figure 7(c). *The* `Node` *object (linked with* `next` *from the root) the programmer was thinking about is now located at a upper-right position from the root* (the dashed oval in the figure). *At the position the programmer was focused on, there is a* `Node` *object referenced by the* `left` *field of the root node* because it is created by the second execution of line 8. Another problem is that a context that differs from the context they were focusing on is specified. They must expect the context of function call of line 10 during an execution of function call of line 14 as a context of the snapshot view mode. However, a specified context after the insertion differs from the expected context because an addition of several functions calls by the insertion changes the focused context of function call from second execution to ninth execution.

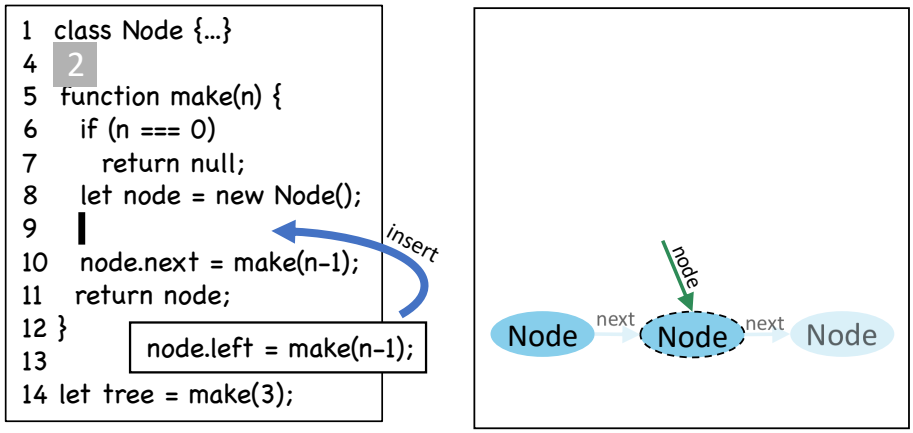
4.3.3 Context-Sensitive Identification for Mental Map Preservation

When a program is modified, Kanon visualizes the new object graph and maintains the context (in the snapshot view mode) so as to preserve the mental map. Here we first explain the requirements for this feature and then describe the proposed mechanism.

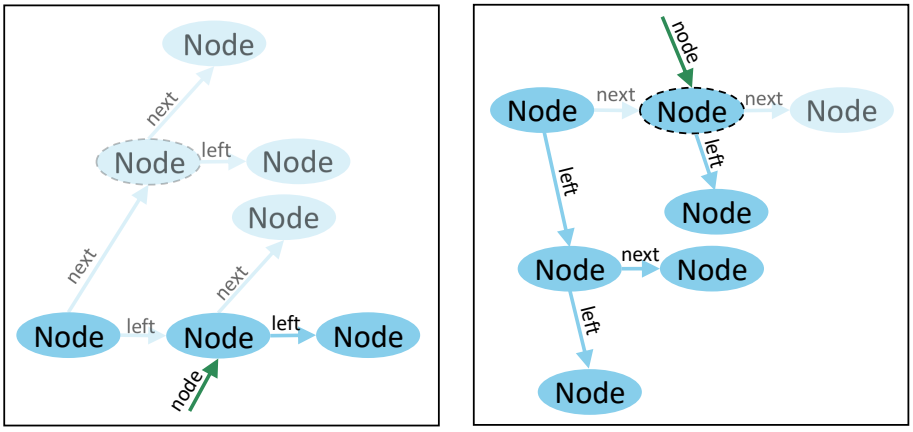
Since Kanon executes the modified program under a fresh environment, it needs to identify (1) a visualization context that corresponds to the one previously displayed, and (2) mapping between objects created in the execution of the modified program and those created in the previous program. in Figure 7, this means (1) identifying one of seven executions of line 8 that corresponds to the second execution in the previous program, and (2) identifying three of seven `Node` objects that correspond to the ones created in the execution of the previous program.

We propose a novel technique, called *calling-context sensitive identification*, for preserving mental map of object graphs[8]. The technique gives a calling-context based identifier, called *context-sensitive ID*, to each function call, and records each object graph by associating the context-sensitive IDs. When it executes a modified program, it selects an object graph matching the context-sensitive ID, and draws nodes

[8]The use of calling-contexts per se is not a novel idea as there are systems that use calling-context for identifying corresponding execution points before and after program modification. The novelty of the paper is the use of calling-context for visualization of object graphs. We discuss the existing systems in Section 6.



(a) An incomplete program that creates a binary tree. **(b)** The visualization before insertion. The dashed node is the programmer's focus.



(c) A naïve visualization after insertion. The position of the dashed node (the `next` of the root) is moved. **(d)** A mental-map-preserved visualization. The dashed node stays at the original position.

Figure 7. Naïve and mental-map-preserved visualizations after code editing. (For the ease of understanding, the nodes that are created beyond the current visualization context are also drawn.)

so that the objects with the same context-sensitive ID will be placed at the same positions.

Figure 8 shows two call graphs that explain the calling-context sensitive identification. Those call graphs (note that they are *not* Kanon's visualization) respectively represent the executions of the programs before and after the insertion. A node of the call graphs is either a function call or object creation, attributed with a label (e.g., `call1` and `new1`) that denotes a source code location. A context-sensitive ID of a call graph node is a list of the labels on the path from the root node.

Kanon uses the context-sensitive IDs, when a program text is modified, to identify the "same" visualization context and to identify the "same" object. In Figure 8, when the current context was the second call to `make` in the older program, the context-sensitive ID of the context is `"call2-call3"`. In the call graph of the modified program, the context that has the same context-sensitive ID is the right child of `make(3)`. This means that the executions triggered by the newly added line are successfully skipped even in the modified program.

In addition, it uses the context-sensitive ID of the `new` expression as the object ID. In Figure 8, the secondly created `Node` object in the older program has `"call2-call1-new1"`. When the program is modified, the object that has the same ID in the new call graph will be placed at the same position.

Our proposal can be summarized in this way:

- We give a unique label to each program location (precisely, we only maintain labels for `new` expressions, object literal, method call expressions, and loops.) We

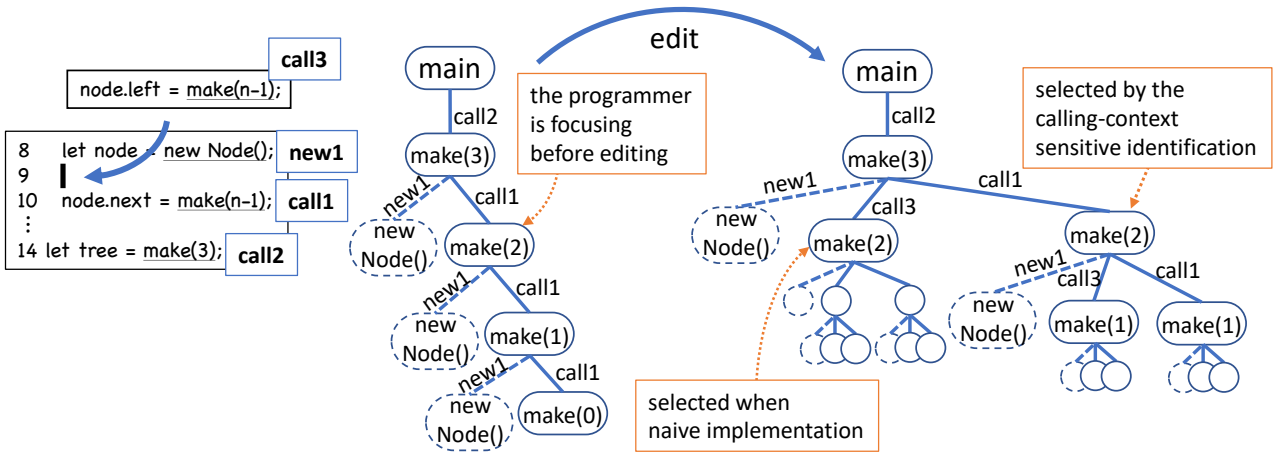


Figure 8. An example of labeling and simplified call trees before and after editing on Figure 7. A new expression is also regarded as a node of the call tree.

preserve the labels by tracking the modification when the programmer edits the program text.

- We give a context-sensitive ID to each execution of an expression such as function call and new expression. In order to determine each object's context-sensitive ID, a stack (each frame of which is configured by either method call label, new expression label or a pair of loop label and loop count) manages the context during execution.
- When new expression is executed, we use the stack information for context-sensitive ID as the ID of the created object.
- When we draw an object graph obtained from an execution of a modified program, we lay it out so that each object will be placed at the same position of the object with the same ID in the execution of the previous program.
- When a program modification changes the call tree, we change the context for snapshot view mode so that the context after the change is the same as the context-sensitive ID before editing.

Intuitively, we consider two objects to be the same when they are created by the same expression, and the execution of the new expression has the same calling context and the same loop count.

Kanon manages a table of program locations for associating expressions with labels, which is robust for most type of editing operations, but has some limitations. When Kanon finds an expression in a program, it gives a new label and records the beginning and ending locations. Upon an editing operation like insertion or deletion of characters, it shifts those locations if necessary. Therefore, it can identify the "same" expressions before and after editing in most cases. There are some operations, such as cut-and-paste and undoing, that the current implementation cannot keep track of, but we believe some of those operations can also be supported by bookkeeping the labels for the text inside the cut- and undo-buffers in the editor.

4.4 Automatic Layout Engine

The current automatic layout engine implemented in Kanon specifies special layout behavior for some structures. Currently, these include binary trees and linked lists. In the case of binary trees, the nodes are specially arranged only if each element is constructed by a Node class and the left element and the right element are represented by `left` and `right` properties. In the case of linked lists, the nodes are specially arranged only if each element is constructed by a Node class and the next element is represented by the `next` property.

In order to implement the above, it is necessary to identify the specified structures from the set of objects. First, in order to find the root of the tree or the head of the list, we must check both sides of each edge. If the root of the tree or the head of the list is found, we then calculate the position of each element. In the case of binary trees, we position each element so that the distance between elements at the deepest level is kept above a threshold. We then set the horizontal position of the parent element to the center of the horizontal position of its child elements. In the case of linked lists, we position each element so that the distance between elements is kept above a threshold. At this time, we adjust the entire graph to preserve the center of gravity of the elements.

However, this layout engine still needs more improvements. The layout engine should recognize arbitrary data structures other than lists and trees. It should also provide a mechanism to shrink or fold unimportant nodes so that the programmer can see a large data structure within a limited drawing area. Supporting customized visualization, where the programmer can control the presentations of data structures, is also important.

4.5 Steadiness

Following the lessons from Hancock and Victor [Hancock, 2003; Victor, 2012], we implemented several mechanisms to stabilize the visualization in Kanon, though there are still many challenges. We here explain those mechanisms and then present the remaining challenges.

Similar to many live programming environment, it keeps the previous visualization when the program has a syntax error. This prevents the visualization flicks while editing the program text.

When the execution of a program causes a runtime error (e.g., null pointer dereferencing), it either updates the visualization if the error happens after the previous visualization context, or keeps the previous visualization with translucent colors[9]. The former case is useful to see the immediate effect of the code fragment being edited, regardless the future errors. Surprisingly, some live programming environment, such as Khan Academy's Live Editor, does not have this feature. For the latter case, an alternative is not hide the visualization. We did not do so because in Javascript programs often cause runtime errors while they are being edited. For example, when we are typing a long variable name, the program causes an unbound variable error at runtime until we finish typing. Making the visualization translucent in the latter case is important to alert the programmer of the error; otherwise the programmer sometimes misinterpret that the program runs up to the previous visualization context yet no changes were made to the objects.

When it updates the visualization with a new snapshot (either selected by cursor movement or by editing the text), it suppresses re-drawing the object graph as long as the graph is topologically equivalent to the shown one. This is a workaround to avoid the inconvenient feature of underlying visualization library, namely vis.js, that randomly changes the geometric positions of edges every time it draws the same graph.

5 Initial Evaluation

As an initial evaluation, we carried out a user experiment with 13 participants. The purpose of this experiment is to collect programmers' opinions about the current implementation of Kanon and to clarify future improvements of Kanon. Due to the limitation of the space, the detailed report on the experiment is written in Appendix A. This section presents a summary of notable findings from the experiment.

In the experiment, each participant is asked to solve two tasks using Kanon and using a simple textual live programming environment (TLPE) after taking a short tutorial session. The first task we gave to the participants, is to define a rotating method for a binary tree. The second task is to create a reverse method for a doubly-linked tree. We then had an interview session to collect the overall impression and the opinions about Kanon's features.

From the participants' opinions, we found the majority of them were positive about Kanon. We also found that some participants drew object diagrams by hand with TLPE, which would indicate necessity of this kind of visualization environment.

As for the time to complete tasks, we could not observe clear difference between two environments. For a simpler task, TLPE seems to be slightly better. Though the time to complete tasks is not our primary interest of the evaluation, we would think that Kanon is as usable as TLPE.

We found that the participants make and handle errors differently with Kanon and TLPE. The first difference is that the time to resolve runtime errors. With TLPE, the participants spent longer time with erroneous states for a task that needs to define a loop. Though we do not have clear explanation for this, one possible reason would be that Kanon helped finding an incorrect state that will cause an error in the subsequent execution.

The second difference is that, with Kanon, several participants took an inappropriate plan to solve the problem. Since the task is to reverse a doubly-linked list, the function must have a loop scanning over the nodes. A correct plan is to let each iteration of the loop modify the forward and backward references of a focused node. However, some participants with Kanon planned to modify the forward reference of the focused node and the backward reference of the next node in one iteration (which was at least not easy to maintain references consistently across iterations, and all of them eventually gave up the plan). Though we cannot investigate the cause of this mistake, we conjecture that the visual representation might mislead the programmer at the planning of a solution. With Kanon, the programmer starts defining `reverse` with visual representation, where the first and the second nodes reference each other. With this visual clue, one might plan to modify those two references.[10]

The third difference is that, with Kanon, several participants took, when using Kanon, took more time to *notice* occurrence of errors. Though this would be mainly because of the way of presenting an error of the current implementation, this would also be due to the policy of our visualization, which keeps showing a previous visual image when an error occurs. This policy is based on the fact that, when the programmer is editing a program, it transiently becomes incorrect either syntactically or semantically. By preserving a previous image, the next image from a successfully executed run is smoothly connected with an animation. At the same

[9] This feature was not available when we carried out the user experiment in Section 5.

[10] The current implementation draws nothing for a field with `null`. Hence the first node has only one outgoing reference. This could also be the cause of the mistake.

time, by seeing the previous image, the participants sometime misunderstood as the program was executed without errors but the object graph was not changed.

6 Related Work

Among the programming environments with data visualization, Python Tutor [Guo, 2013] has a 'live' feature. When a program is edited, it automatically re-executes the program and updates the visual representations. However, there are several limitations. (1) There is no synchronization mechanism that automatically selects a program state at the cursor position for visualization. Hence the programmer needs to manually seek the point of visualization by using the forward and backward buttons. (2) When a program is edited, the execution point for visualization can be shifted to an unexpected point. As far as the authors observed, when the system is showing at the N-th step a program, it will show, when the program is changed, at N-th step of the modified program. Therefore, a change to the code that has been executed until the N-th step can lead the system to an unexpected point of execution. (3) It does not have a mental map preservation mechanism when a program is edited. Instead, when a program is edited, it redraws a newly obtain object graph without using past information.

The Morphic environment in Self [Ungar and Smith, 1987] lively integrates the code editor and the object inspector, which displays objects as a node-link diagram. The selection of visualized objects is manual. It always displays the *current* state of objects; i.e., it is not possible to show the paste state in an execution without resorting to a breakpoint debugger.

The calling-context sensitive identification technique (Section 4.3) can be used for problems that need to align executions of two versions of a program. For example, example-centric programming [Edwards, 2004] uses a similar technique to identify a context of a test case that is created for an older version of a program.

Zimmermann and Zeller propose to display a visual object graph in a debugger [Zimmermann and Zeller, 2002]. They also propose to highlight the difference between two object graphs. Their approach calculates a *correspondence graph* to determine the same objects.

Back-in-time debuggers [Lewis, 2003] can show a state of a program at an arbitrary time in the execution. Similar to Kanon, they record program execution through program instrumentation. However, research in debuggers mainly focuses on recording more detailed information than mere object graphs. It is not clear if those techniques can be directly applied to live programming environments, where programs are frequently modified.

Kanon assumes that the programmer writes a program in a test-driven development [Beck, 2003] style. Such a style can be commonly found in live programming [Imai et al., 2015; Victor, 2012] as well as in example-centric programming [Edwards, 2004].

7 Conclusion

We propose Kanon, a live data structure programming environment. The notable feature is the mechanism that helps to connect the visual representation and the program code. The snapshot view mode shows the object graph when the program execution is reached at the cursor position, which should be relevant to the programmer's mental state. The summarized view mode shows the effect of an expression under the cursor throughout the execution. This helps the programmer to check whether the current expression behaves as expected. The jump-to-construction mechanism helps to connect a graphical element to an element in a program. We proposed *the calling-context sensitive object identification technique* to preserve mental map of representation between the graphs generated by modified programs. Finally, we carried out qualitative user experiment and it was found that Kanon could help programmer imagine data structures and most of the participants have positive impression. However, Kanon still has some improvements, further improvements will make data structure programming easier and more enjoyable.

Future work includes the development of a better object graph layout algorithm that is closer to the programmer's expectations, improvement of feedback performance, a selective visualization mechanism for larger object graphs, more efficient display and selection method for the calling-context, and introduction of direct manipulation mechanism.

A Initial Evaluation

As an initial evaluation, we carried out a user experiment[11] in order to collect programmers' opinions about the current implementation of Kanon. Since we have not yet implemented many practical features such as code completion, we do not believe that we can do the meaningful quantitative evaluation. (In the experiment, we measured time to task completion, which is not a primary purpose of the experiment.) Nevertheless, as we will see in our experiment, we observed interesting programming behaviors with Kanon, positive opinions on the Kanon's features, and several future improvements.

A.1 Design of the Experiment

In our experiment, we let the participants use Kanon to solve several programming tasks in order to observe their usage of the Kanon's features, and to gather their opinions. In addition, we designed the experiment with the following questions in our mind.

[11] This experiment was carried out in Japanese all the time. In this paper, the participants' opinions have been translated into English.

- *Do graphical outputs make difference in programmer's behavior from textual outputs?*
  We build a textual live programming environment (called TLPE hereafter) as a counterpart of Kanon and let the participants use both environments.
- *Does the amount of programming experience affect the usage of Kanon?*
  Our experiment had 13 participants consisting of 9 students (at the senior undergraduate and graduate levels) and 4 computer scientists in a corporate research laboratory. Note that all the students and one scientist have heard about Kanon before the experiment, but none of them have ever used it.
- *Does difficulty of programming tasks affect the usability of Kanon?*
  We prepared two tasks with different difficulties and let the participants solve them. The simple one merely requires to modify a few object references. The difficult one requires to traverse references while modifying the references themselves.

A.2 Experimental Procedure

We carried out the experiment for each participant one by one. The participant took the four phases, namely tutorial, practice, main and interview, which amount to approximately one and a half hours in total. We recorded the participant's activity by recording the computer's screen and participants' voice. Throughout the experiment, we asked the participants to speak out their thoughts, for example "I'm confused now", "Why is the figure displayed like this?" and "Oh, this program includes an error." What the participants are thinking tells us where they are paying attention to during programming.

A.2.1 Tutorial

In the tutorial phase, the participant is asked to read a 67-pages document that describes the usage of Kanon and the format of the tasks. In this experiment, all the tasks are to define a method for a common data structure. The participants were given the definition of the data structure, a definition of the method without an empty body, and a series of method call expressions that serve as test cases, which cover all the situations. The document uses the scene in which a programmer defines the `LinkedList.add` method as an example. At the same time that the participants read the document, they are allowed to use Kanon to grasp how Kanon works.

A.2.2 Practice Phase

In the practice phase, the participant is asked to define a `LinkedList.insert` method as the exercise simple task using Kanon in order to get used to Kanon and the format of tasks.The excercise task took up to approximately 15 minutes. Throughout this phase, we allowed the participants to question anything. After they have completed the task or time is over, we commented the feedback and the answer to the task to them.

A.2.3 Main Phase

In the main phase, the participant tasked to solve two tasks, namely **rotate** and **reverse**, in this order. We grouped the participants into two, and assigned Kanon or TLPE to those tasks according to Table 2. Each task was given a 20 minutes time limit.

The **rotate** and **reverse** tasks are to define a method that rotates a root node of binary tree, and a method that reverses a doubly-linked list, respectively. The former task can be accomplished by merely modifying a few references in the given tree nodes. The latter task is rather difficult, as it requires to follow links between nodes while modifying those links.

TLPE is a simple live programming environment as shown in Figure 9. It provides a customized `println` function that prints out data on the right-hand side of the screen. It is live in the sense that the output is immediately updated whenever the code on the left-hand side changes, which is similar to Khan Academy's Live Editor [Resig, 2012] and YinYang [McDirmid, 2013]. Unlike the built-in `print` function in JavaScript, the `println` function displays internal elements in a nested data structures. It also supports cyclic structures (by showing #*n* as in the figure).

A.2.4 Interview

In the interview phase, one of the authors asked the participants several questions. The first question is about difficulties throughout the experiment. The role of this question is to clearly remind the participants of their thoughts during the experiment. Then, for each feature of Kanon, we asked the participants their opinion. We encouraged them to answer, not just "good" or "bad", but rather concrete opinions on specific parts of the feature, and possible improvements. Finally, we asked an overall impression of Kanon. The participants may answer the impression of Kanon itself, or in comparison with TLPE.

A.3 Results

Table 1 and Table 2 show a quantitative result of this experiment such as time taken to complete the tasks and a count of using features of Kanon. The qualitative opinions received in the interview are described below.

A.4 Opinions about Kanon's Features

About **the snapshot view**, there are several positive opinions like:

- *"The feature was very helpful for changing references"*, and

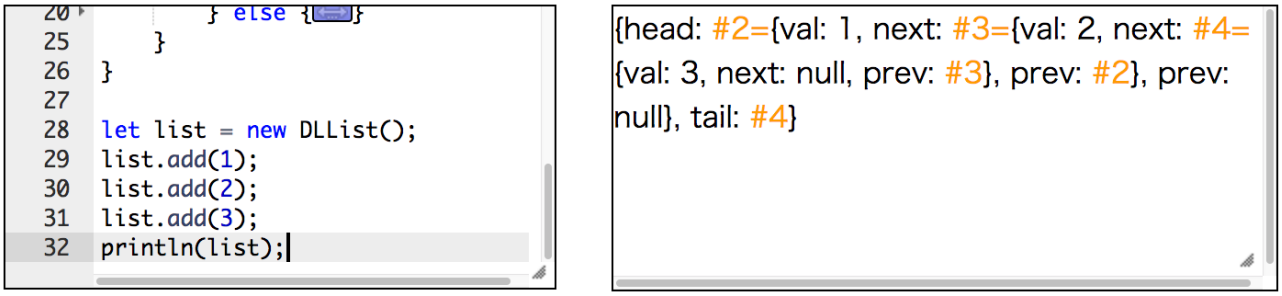


Figure 9. Textual Live Programming Environment (TLPE).

Table 1. Average number of times of uses of each feature. ("Snap:Summ" means the proportion of each in the overall task. The column of "Context" is constructed the number of (correct usage : incorrect usage). Each column of "JtoC" and "print" is the number of using *jump to construction* and print statement, respectively.)

(a) task1

Snap : Summ	Context	print	JtoC
80.6% : 19.4%	2.57 : 3.57	2.17	0.29

(b) task2

Snap : Summ	Context	print	JtoC
96.9% : 3.1%	7.5 : 3.17	6.29	0.17

Table 2. The participant information.

	Group A	Group B	total	min/ave/max exp	#js
task1 (**rotate**)	Kanon	TLPE			
task2 (**reverse**)	TLPE	Kanon			
#Students	5	4	9	2 / 4.4 / 10	2
#Researchers	2	2	4	14 / 17 / 25	1
#total	7	6	13	2 / 8.3 / 25	3
min/ave/max exp	2 / 7.3 / 14	3 / 9.5 / 25	2 / 8.3 / 25		
#js	3	0	3		

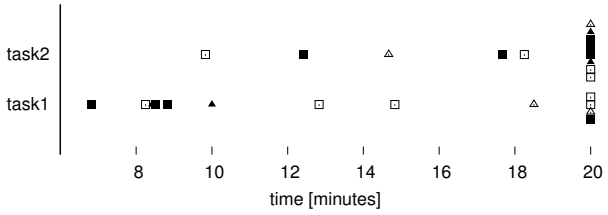


Figure 10. Time taken to solve the tasks.

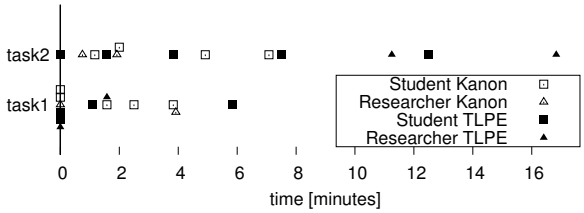


Figure 11. Error time.

- *"The feature is reasonable because we often want to see the states (of the program) around the code fragment being written."*

Visualization of variable references (e.g., the arrows labeled `this` and `temp` in Figure 3) are also positively taken by most participants, but also had suggestions like:

- *"I also wanted to see arrows for function arguments*[12] *when I was writing a recursive function,"* and
- *"It looks strange that the arrows for variables lacks originating ovals."*

The summarized view was not used by the most participants. One participant gave the reason:

- *"I only needed the snapshot view mode".*

However, there are opinions that suggest its potential like:

- *"The visualization with orange and green arrows was very easy to recognize,"* and *"I used it to show the final*

[12]The current implementation of the *snapshot view* merely displays locally declared variables and `this`, but not function arguments.

state of the program, and completed the task by imagining the other states," (This participant mainly used the summarized view mode.)
- *"It might be useful when it is hard to understand an overview of a program"*, and
- *"It might be useful for tasks of fixing bugs."*

A few participants used the **jump-to-construction** feature but they thought that the feature was not so helpful. The opinions are:

- *"I had no chance to use it,"*
- *"It might be useful for larger programs because it would be difficult to understand the overview,"* and
- *"Construction sites are not so relevant when we modify fields in existing objects."*

The **automatic layout engine** had positive comments like:

- *"The visualization was similar to what I imagined,"* and
- *"The figure after the completion of the task was cleanly arranged without moving the node ourselves."*

At the same time, it also had suggestions like:

- *"Though the final layout looks good, it does not in the middle of programming,"* and
- *"I wanted to undo the (automatic) layout as it became messy."*

A.5 Overall Impression

Overall, the participants gave positive comments like:

- *"It is amazing. I want to use it,"*
- *"With visualization, it was easy to program since I often draw pictures when I reason about data structures,"* and
- *"I felt it is wonderful when I was solving a task with TLPE."*

and also several suggestions for future improvements like:

- *"It was helpful for those tasks, but not sure if it will be so for other situations and for large programs,"*
- *"When the object graph disappears, I wanted to know the reason*[13]*,"* and
- *"After I thought about (the strategy) based on the visualization, I had to think again based on the program. I wish I could generate code fragments by directly manipulating the object graph. "*

A.6 Discussion

This section discusses finding in the results of the experiment as well as the observations of the participants' behavior by the authors.

[13]In the version we used for the experiment, Kanon will erase the object graph when a runtime error occurs. The error message was actually displayed bottom of the screen.

A.6.1 Kanon vs. TLPE

The experiment did not give a clear answer whether the graphical representation as opposed to the textual representation is useful. With respect to the task completion times on Figure 10, TLPE is faster than Kanon for the `rotate` task, or is as fast as Kanon for the `reverse` task. (Again, due to the limitations of the current implementation, we do not consider the task completion times are the primary factor of the experiment. Also, the number of participants is not large enough to evaluate statistical significance of those figures.)

From the closer observations, we had the following findings and insights.

- Roughly the half of the participants misused the Kanon's features related to changing specified context. As the "Context" column on Table 1 shows, selection of the execution context seems to be difficult. This suggests that Kanon needs more improvements on its GUI.
- Some participants, when they were using TLPE, drew object graphs on a paper. This suggests usefulness of graphical representation regardless the environment used.
- The styles of problem solving are different by the participants. Some carefully thought out algorithms before writing code, and some others carried out the trial-and-error style programming. Those difference in the styles and the difference programming environment could affect each other, which should be investigated in future.
- We observed interesting difference in the participants' behaviors when errors occurred. First, from the observations of the behaviors, we found that many participants took, when using Kanon, longer time to notice occurrence of errors. This would be partly because the current design of the environment that reports errors as a plain text at the bottom of the screen, which hardly attract the programmer's attention. However, this would also due to the policy of our visualization, which keeps showing a previous visual image when an error occurs. This policy is based on the fact that, when the programmer is editing a program, it transiently becomes incorrect either syntactically or semantically. By preserving a previous image, the next image from a successfully executed run is smoothly connected with an animation. At the same time, by seeing the previous image, the participants sometime misunderstood as the program was executed without errors but the object graph was not changed. With TLPE, the programmers can immediately notice runtime errors because an error will prevent execution of subsequent `println` calls, which results in disappearance of the output.
  Second, for the `reverse` task, the participants with TLPE spent longer time with erroneous states. Though

we do not have clear explanation for this, one possible reason would be that Kanon helped finding an incorrect state that will cause an error in the subsequent execution.
- Even though we do not observe clear difference in the task completion times, the participants opinions favor Kanon as reported in Section A.5. We would like to consider the reasons why they thought like that.

A.6.2 Students vs. Researchers

With respect to the task completion times, we did not observe clear differences between inexperienced and experienced participants (i.e., the students and the research laboratory scientists, respectively). This might be because the students took the courses on programming and data structures more recently.

We noticed difference a difference between students and researchers in their programming styles. While the researchers tried to add more test cases on top of the provided cases, the students declare completion by only considering the provided test cases.

A.6.3 Difficulty of the Tasks

Difficulty of tasks and visualization can affect the types of mistakes that the programmer makes. While the types of the mistakes with Kanon and TLPE are not different for the easier task (**rotate**), we observed a unique kind of mistakes with Kanon for the more difficult task (**reverse**).

For the **reverse** task, several participants with Kanon took an inappropriate plan to solve the problem. Since the task is to reverse a doubly-linked list, the function must have a loop scanning over the nodes. A common and correct plan is to let each iteration of the loop modify the forward and backward references of a focused node. However, some participants with Kanon planned to modify the forward reference of the focused node and the backward reference of the next node in one iteration (which was at least not easy to maintain references consistently across iterations, and all of them eventually gave up the plan).

Though we cannot investigate the cause of this mistake, we conjecture that the visual representation might mislead the programmer at the planning of a solution. With Kanon, the programmer starts defining `reverse` with visual representation, where the first and the second nodes reference each other. With this visual clue, one might plan to modify those two references.[14]

In general, we believe that making a correct plan is crucial for solving difficult programming tasks regardless the programming environment used. With a new type of programming environment, we would probably need more experience to develop good recipes for typical types of problems.

A.6.4 Is Kanon Helpful?

In the experiment, we observed rare usage of some features, namely the *summarized view mode* and *jump-to-construction*. We presume that this is due to the type of the tasks used in our experiment, which are to define a new function body. We design the *summarized view mode* and *jump-to-construction* for the situations of modifying a program and of understanding program behavior, respectively. We might have observed more usage with those features if the experiment included such tasks.

As mentioned in Section A.6.1, notifying the programmer an error as early as possible is crucial. We found that the problem is not trivial. In a live programming environment, a program transiently becomes an erroneous state when the programmer edits a code fragment. The environment should also delay notification so as to smoothly connect visual images between the states without errors. The problem is even more difficult when a program correctly runs around the code being edited, but causes an error at the later execution point.

[14] The current implementation draws nothing for a field with `null`. Hence the first node has only one outgoing reference. This could also be the cause of the mistake.

Acknowledgments

The authors would like to thank Tomoki Imai for his advice on the implementation techniques, Jun Kato for his valuable comments, and Sean McDirmid for shepherding the paper through the review process. We also thank all the user experiment participants. This work was supported by JSPS KAKENHI Grant Numbers 26330078 and 18H03219.

References

Samuel Aaron and Alan F. Blackwell. 2013. From Sonic Pi to Overtone: Creative Musical Experiences with Domain-specific and Functional Languages. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design (FARM '13)*. ACM, New York, NY, USA, 35–46.

Apple Computer. 2016. Swift Playgrounds. http://www.apple.com/swift/playgrounds/. Accessed February 2017.

D. Archambault and H. C. Purchase. 2012. The Mental Map and Memorability in Dynamic Graphs. In *2012 IEEE Pacific Visualization Symposium*. 89–96.

Kent Beck. 2003. *Test-Driven Development: by Example*. Addison-Wesley Professional.

M. H. Brown. 1991. Zeus: a system for algorithm animation and multi-view editing. In *Proceedings 1991 IEEE Workshop on Visual Languages*. 4–9.

Marc H. Brown and Robert Sedgewick. 1984. A System for Algorithm Animation. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '84)*. ACM, New York, NY, USA, 177–186.

Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. 2013. It's Alive! Continuous Feedback in UI Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 95–104.

Almende B.V. 2018. vis.js - A dynamic, browser based visualization library. Retrieved April 23, 2018 from http://visjs.org/

Jonathan Edwards. 2004. Example Centric Programming. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, Doug Schmidt (Ed.). ACM, New York, NY, USA, 84–91.

Chris Granger. 2012. Light Table. http://www.chris-granger.com/lighttable/. Accessed February 2017.

Philip J Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for CS Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. ACM, 579–584.

Christopher Michael Hancock. 2003. *Real-time Programming and the Big Ideas of Computational Literacy*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA.

T. Dean Hendrix, James H. Cross, II, and Larry A. Barowski. 2004. An Extensible Framework for Providing Dynamic Data Structure Visualizations in a Lightweight IDE. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04)*. ACM, New York, NY, USA, 387–391.

Ariya Hidayat. 2018. Esprima. Retrieved April 23, 2018 from http://esprima.org

Tomoki Imai, Hidehiko Masuhara, and Tomoyuki Aotani. 2015. Making Live Programming Practical by Bridging the Gap Between Trial-and-error Development and Unit Testing. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity* (2015-10-25), Jonathan Aldrich (Ed.). ACM, ACM, 11–12.

Fabian Jakobs. 2018. Ace - The High Performance Code Editor for the Web. Retrieved April 23, 2018 from https://ace.c9.io

Saketh Kasibatla and Alessandro Warth. 2018. Seymour: Live Programming for the Classroom. Retrieved June 17, 2018 from https://harc.github.io/seymour-live2017/

Jun Kato, Sean McDirmid, and Xiang Cao. 2012. DejaVu: Integrated Support for Developing Interactive Camera-based Programs. In *Proceedings of the 25th annual ACM symposium on User Interface Software and Technology (UIST'12)*. ACM, 189–196.

Khan Academy. 2018. Intro to JS: Drawing & Animation. https://www.khanacademy.org/computing/computer-programming/programming. Accessed February 2017.

Yi-Yi Lee, Chun-Cheng Lin, and Hsu-Chun Yen. 2006. Mental Map Preserving Graph Drawing Using Simulated Annealing. In *Proceedings of the 2006 Asia-Pacific Symposium on Information Visualisation - Volume 60 (APVis '06)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 179–188. http://dl.acm.org/citation.cfm?id=1151903.1151930

Bil Lewis. 2003. Debugging Backwards in Time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*. arXiv:cs.SE/0310016 http://arxiv.org/abs/cs.SE/0310016

Henry Lieberman and Christopher Fry. 1995. Bridging the Gulf Between Code and Behavior in Programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '95)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 480–486.

Sean McDirmid. 2007. Living it Up with a Live Programming Language. In *In Proceedings of Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. ACM, 623–638.

Sean McDirmid. 2013. Usable Live Programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, New York, NY, USA, 53–62.

Akio Oka, Hidehiko Masuhara, and Tomoyuki Aotani. 2017a. The Visualization of Data Structures and Interactive Features for Live Programming. In *The 113th IPSJ Workshop on Programming*.

Akio Oka, Hidehiko Masuhara, Tomoki Imai, and Tomoyuki Aotani. 2017b. Kanon: Data Structure Programming using Live Programming Environment. In *Proceedings of the 10th JSSST Workshop on Programming and Programming Languages (PPL 2017)*.

Akio Oka, Hidehiko Masuhara, Tomoki Imai, and Tomoyuki Aotani. 2017c. Live Data Structure Programming. In *Companion to the First International Conference on the Art, Science and Engineering of Programming (Programming '17)*. ACM, New York, NY, USA, Article 26, 7 pages.

John Resig. 2012. Redefining the Introduction to Computer Science. http://ejohn.org/blog/introducing-khan-cs/. Accessed March 2018.

John T Stasko. 1989. *TANGO: A Framework and System for Algorithm Animation*. Technical Report. Providence, RI, USA.

David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87) (ACM SIGPLAN Notices)*, Norman Meyrowitz (Ed.), Vol. 22(12). ACM, ACM, Orlando, FL, 227–242.

Bret Victor. 2012. Inventing on Principle. Keynote Talk at the Canadian University Software Engineering Conference (CUSEC).

Thomas Zimmermann and Andreas Zeller. 2002. Visualizing Memory Graphs. In *Software Visualization*. Springer, 191–204.