Causette: User-Controlled Rearrangement of Causal Constructs in a Code Editor

Alice Martin, Mathieu Magnaudet, Stéphane Conversy

To cite this version:

Alice Martin, Mathieu Magnaudet, Stéphane Conversy. Causette: User-Controlled Rearrangement of Causal Constructs in a Code Editor. ICPC '22: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, IEEE/ACM, May 2022, Pittsburgh, United States. 10.1145/3524610.3527885 . hal-03659579

HAL Id: hal-03659579
https://enac.hal.science/hal-03659579

Submitted on 5 May 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Causette: User-Controlled Rearrangement of Causal Constructs in a Code Editor

Alice Martin, Mathieu Magnaudet, Stéphane Conversy

first.last@enac.fr

ENAC, Université de Toulouse

France

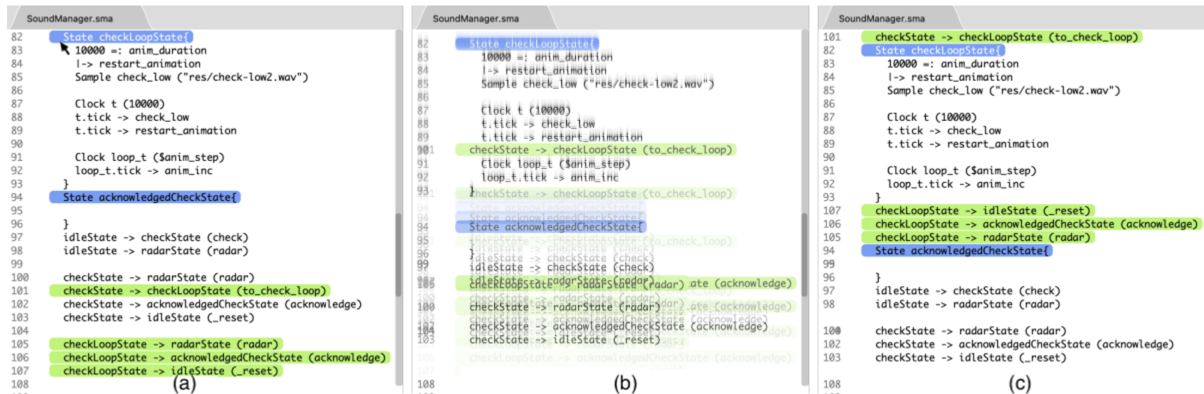

Figure 1: *Interaction 2.* Reordering FSMs: (a) The user hovers over on a declaration of a state (blue) which highlights the in and out transitions (green); (b) the user clicks on the state, and the system animates a rearrangement of the transitions; (c) the new y-ordering of the statements matches the causal relationships being analyzed (in-transitions above state above out-transitions).

ABSTRACT

Programming interaction usually involves specifying causal relationships such as input events triggering a state change or the propagation of values. Such code may reside in several locations and its execution is driven by multiple causal chains, which hinders the programmer's ability to understand and fix it. We designed Causette, a set of four novel interaction techniques for a code editor. They consist in rearranging causal constructs on demand to make the code representation consistent with the causal chain being analyzed by the user. We ran an experiment showing that Causette may be more usable than a regular editor for some code understanding tasks. This work suggests that rearranging interaction code may help developers better understand and fix it.

CCS CONCEPTS

• **Human-centered computing → Visualization toolkits**; **User interface programming**; • **Software and its engineering → Development frameworks and environments**.

KEYWORDS

Interaction programming, code visualization, causality

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

ICPC '22, May 16–17, 2022, Virtual Event, USA

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9298-3/22/05…$15.00

https://doi.org/10.1145/3524610.3527885

ACM Reference Format:

Alice Martin, Mathieu Magnaudet, and Stéphane Conversy. 2022. *Causette*: User-Controlled Rearrangement of Causal Constructs in a Code Editor. In *30th International Conference on Program Comprehension (ICPC '22),May 16–17, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524610.3527885

1 INTRODUCTION

Programmers of interactive systems e.g. Graphical User Interfaces that react to input events from the keyboard, the mouse, or the touchscreen, face specific challenges due to the interrelated behavior of numerous components and uncontrolled, unpredictable flow of external events [37]. A number of textual programming languages provide dedicated language constructs to tackle the complexity of writing interactive programs. Some of them propose reactive constructs that avoid the need to write code for updating outputs when the inputs of a computation change [64]. Others use traditional object-oriented constructs to describe state-based behavior [1]. Conceptual frameworks, syntaxes and tools have been proposed for user interface development [49], natural programming [50], interaction-oriented programming [40], or web programming [4]. Still, most approaches rely on code split over different source files, each dedicated to the description of interactive behavior at a local level. This breakdown hinders the programmer's ability to understand interactive behavior. As a consequence, interaction code is hard to debug and maintain.

The problem: understanding causality

We believe that developers should be able to follow and understand the *causal chains* to assess whether the code of an interaction will behave correctly. Following the philosopher of science W. Salmon [57], we take *causality* as a spatio-temporal process involving the transmission of "information, structure and causal influence". We consider that two processes A and B have a *causal relationship* if A always precedes B and B occurs every time A occurs. In other words, "causality" means that the execution of a piece of code follows the occurrence of an input event. *Causal chains* are a set of elementary causal relationships that form a chain.

General-purpose and interaction-oriented programming languages provide programmers with "causal programming constructs" i.e. syntactic expressions that establish a particular causality between two pieces of code. Some of Java Swing, Qt, SwingStates [2] or QML APIs and syntaxes can be considered as causal constructs. With Java Swing, the registering of a listener callback is a causal construct expressed with `source.addListener(listener)`. With Qt, the connection between a signal and a slot is expressed with `object1.signal.connect(object2.slot)`. With SwingStates, the transition between two states is expressed with `Transition t = new PressOnShape(BUTTON1, "»menuOn")` while with QML a transition is expressed with `DSM.SignalTransition {targetState:finalState signal:button.clicked}`. In Smala[40], the binding between the activation of two processes is expressed with `p1->p2`, an assignment with `p1=:p2`, a dataflow with `p1=>p2` and a transition with `s1->s2(event)`.

Such causal constructs may have been defined by a programmer with the same source or destination, over multiple code locations. They thus implicitly form a "causal chain". The challenge we tackled here is the programmer's ability to understand such causal chains. Since interaction is by nature dependent on external events that might occur in any order, understanding an FSM requires taking into account multiple causal chains. Similarly, to understand a dataflow of connected components, developers must follow the flow of data across multiple code locations. Few tools have addressed these difficulties in past work.

Causette

Our goal is to better support programmers when they seek to understand the causal chains involved in interactive programs. Here we introduce *Causette*[1], a set of interaction techniques and representations of textual code. With Causette, a programmer is notably able to bring together the causal relationships that are far away from each other in the source code, with a visual ordering consistent with the conceptual causal order (see fig. 1). Our contributions are:

(1) The elicitation of four design principles (4.2) we relied upon to design four interactions (4) that rearrange interaction code representations to support some interaction programming tasks;
(2) The demonstration of the interactions in relevant scenarios (4);
(3) The evaluation of the interactions with a semi-controlled experiment with 12 professional programmers (5).

[1]Causette is a pun from the French given name Cosette spelled like Causality, and evoking "*-et" HCI terms such as "widget" or "applet".

2 RELATED WORK

We first introduce the terminology we use. As much as functional programming is about programming functions that run within a program that compute results [35], interaction programming is about programming interactions that run within an interactive program i.e. that reacts to external events such as human input or network data reception. Interaction code is the code devoted to the description of interactive programs. Similarly, as much as object-oriented programming [31] languages have been designed to suit the programming of "objects", interaction-oriented [37] programming languages and frameworks [14, 37, 48] have been designed to suit the programming of "interactions". Causette is informed by research in three areas: interaction programming and causality, augmenting textual code, and code representation and animation.

2.1 Interaction programming and causality

In the field of interaction-oriented programming, the causal challenges related to interaction code have been pointed out [14, 37, 47, 48], but few tools have addressed it. The WhyLine debugger [32, 33] provides developers with answers to a *why?*-question regarding a phenomenon they perceive on a user interface i.e. a causality question. Outside the scope of interaction programming, many papers have studied developers' needs, particularly when debugging [36, 61, 63] and proposed new debugging methods, notably in the field of functional reactive programming [58, 59]. However, few of them have been focusing on the debugging of interaction-oriented programs or causal relationships. Aside the field of programming, some work in information visualization have studied the representation of causality. This includes the exploration of graphs and diagrams to visualize causality [22, 23, 62], applied to statistics or to the modelling of distributed systems. However, typical graph layouts may not be appropriate to follow the control flow of an application, as they force the readers to visually hop from node to node in arbitrary directions. Those limits have been addressed by tools like DA4Java [53] for Java source code. To make the graph more readable, a set of features allows to incrementally compose graphs and remove irrelevant nodes and edges from graphs.

2.2 Augmenting textual code in IDEs

Several issues with program understanding motivate work on textual code augmentation. One issue is related to file-based IDEs. The literature points out how IDEs lack effective support to browse complex relationships between source code elements. Developers are often forced to exploit multiple user interface components at the same time [34], making the IDE "chaotic" [43]. To prevent time-consuming navigation between files, CodeBubbles [7, 8, 55, 56] offers an integrated development environment for Java. With CodeBubbles, programmers can build working sets composed of code fragments (like methods, small classes, notes, documentation, etc.), displayed in a separate bubble or lightweight window. Programmers can rearrange the layout of the bubbles to create a logical context. VSCode offers a functionality called CodeLens [42]: an actionable contextual information interspersed (from different files) in the edited code. It differs from Causette in terms of interaction technique (pop-up window), type of information (e.g. editing history) and targeted code (imperative code). In the same line of thought, to

overcome the limits of navigation in IDEs, Hunter [19] is a tool for the visualization of JavaScript applications that provides a set of coordinated views that include a node-link diagram that depicts the dependencies among the components of a system, and a treemap that helps programmers to orientate when navigating its structure.

We used a different strategy to bring related chunks of code closer, by inserting "remote" code into the currently edited one. Similarly, "code portals" [9] embed various types of context information uniformly into the main source code view in proximity to the relevant source code. "Fluid source code views" [18] consists of insertions of relevant remote lines of code in the edited file. The aim is to provide the programmer with the control and data flow directly in the edited code, and thereby minimizing navigation. But it is applied to object-oriented programming and relies on displaying hierarchies of methods. SimplyHover [29] is a plug-in for Eclipse that brings the "if" condition next to its "else" counterpart.

Theseus is an IDE extension that visualizes the run-time behavior of a program within a code editor by displaying real-time information about how the code actually behaves during execution [39]. It provides the programmer with the number of calls of a particular function and a collapsible tree of calls. Like Causette, Theseus augments the textual code being edited, by displaying the number of calls to the left of the function header. By contrast, we target another type of information: the causal activation chain. We also focused on state activations in an FSM, but used a quantitative representation of activation recency instead of a numerical representation of the number of calls. We took Theseus evaluation method as inspiration for the evaluation of Causette.

2.3 Code representation and animation

Causette builds upon previous work about graphical code representations. For example, InterState [51] is a programming language and environment that supports developers in writing and reusing user interface code. InterState mixes texts and graphics to represent interactive behaviors using a combination of FSMs and constraints. It also provides programmers with a visual notation to facilitate code navigation and understanding. SwingStates [2] makes a clever use of Java anonymous inner classes to describe FSMs. However, the specification of the interactions is done at a local level, and SwingStates does not offer support for understanding causal chains.

From a more theoretical standpoint, source code shares a lot with a text to be read [54]. There is a distinction made in the literature between textual and visual information. But it is not clear-cut and it is not obvious to favor the latter or the former. Visual programs can even be harder to read than textual programs [26]. The cognitive dimensions of notation is a framework that helps designers analyze interactive tools, including programming environments and languages, may they be textual or graphical [6]. The Physics of Notations framework focuses on the properties of graphical notations [44]. Unifying textual and visual languages shows that both types have much in common and that both should rely on the capability of the human visual system [17]. More recent work on code reading is relevant to Causette. There have been studies on the way programmers read natural language text and code. Some results indicate that code reading is less linear [12, 52] than prose reading because programmer focus on the program execution flow, or that code regularity (same structures repeated time after time) reduces code reading complexity [30]. The effect of ordering on comprehension has been studied [30], with a focus on the ordering of methods. It motivates design principles for Causette, in the sense that it invites to support the matching between linearity of reading order and the readability of the execution flow.

An environment and language suitable for programming should allow one to "follow the flow" and "see the state" [10]. The author designed several interaction techniques and representations to support those two concerns. However, they involved following an imperative flow and data states, while Causette targets the interaction flow or the interactive state. The author claims that the features of the environment are less important than the particular ways of thinking they support. This is what we strive to do for interaction programming: provide programmers with interactions and representations to better apprehend causal relationships.

To augment and represent textual code efficiently, we used animations. Text animations are useful to understand changes of information display [16]. Gliimpse [21] or Diffamation [15] share with our work the use of animations in code: they offer animated transitions to different parts of text (latex markup code and rendered document in Gliimpse, revision history of textual documents in Diffamation). Gliimpse allows users to check and navigate through the code, without leaving the text editor. However, Gliimpse animates from code to rendering, and not from code to code. Finally, animations of word-scale graphics within texts also enable to follow the rearrangement of graphics and make them easier to compare thanks to a vertical alignment [25].

3 THE SMALA LANGUAGE

As mentioned in the related work, the problem of describing and understanding interaction code has been partly addressed with the design of specific, interaction-oriented textual languages. We used such a language, Smala, in the remaining of the paper, as it eliminates some interaction programming problems, like the "spaghetti of call-backs" [46]. Our interaction techniques were designed for this language, to overcome the remaining difficulties (the techniques may be applied to other languages or toolkits such as Qt). This section briefly describes Smala to help the reader understand the code examples.

Smala is a textual, interaction-oriented programming language dedicated to the development of highly interactive software. It takes inspiration from classical reactive languages such as Lustre [27] or Esterel [3], notably adding a specific syntax and a seamless integration of interactive graphics.

At the conceptual level, a Smala program is a declarative specification of a graph of coupled processes going from a set of event sources to a set of output processes. Such specification is akin to the way causal chains are declared in an imperative language/framework such as Java or Qt: adding listeners (Java), or connecting signals to slots (Qt) is a way to declare a causal chain during the *initialization* of an interactive program. The execution is triggered by the occurrence of events that are propagated by an activation vector, resulting from the sorting of the graph of processes.

At the syntactic level, Smala provides specific operators to define causal links between processes (listing 1). For example, the arrow

between two processes specifies that each time the left process is activated then the connected process must be activated (binding). More complex control structures exist such as FSMs. An FSM consists in declarations of states, followed by declarations of transitions (listing 2). Each state may include other processes.

The SMALA syntax has been designed to facilitate the development of interactive software. Indeed, the arrow-based notation is a representation of the control flow of an interactive software. The language designers have also reversed the direction of the classical APL assignment operator (*src* =: *dst* instead of *dst* := *src*) to make it consistent with the directions of other types of flow (binding: − > and data-flow/connector: =>). SMALA makes it easy to combine data-flows and FSMs: a data-flow can trigger an FSM transition and an FSM can start or stop a data-flow (as envisioned in [13, 28]).

However, as with Java or C++/Python/Qt programs, the entire causal chain specification can be spread over several files, which makes it difficult to understand. The SMALA FSM also exhibits some scaling issues with a large amount of states and transitions, and with nested FSMs. Even if we believe that dedicated syntax of SMALA is more usable for interaction programming, it is not enough to understand causality.

Listing 1: Binding & Dataflow

```
Clock cl (500)
Incr inc
cl.tick -> inc.step
//      ^^ binding
// inc is activated every
      500ms

Double v (0)
inc.state => v
//        ^^ dataflow
// v receives inc value
// every time inc is
      incremented
```

Listing 2: FSM

```
FillColor f(0,0,0)
Rectangle r(50, 50, 100, 70)
FSM simple_FSM {
  State idle {
    #000000 =: f.value
  }
  State hover {
    #a0a0a0 =: f.value
  }
  State pressed {
    #ff0000 =: f.value
  }
  idle->hover (r.enter)
  hover->idle (r.leave)
  hover->pressed (r.press)
  pressed->hover (r.release)
  //      ^^ transition
}
```

4 INTERACTION TECHNIQUES

We devised from the state of the art three requirements for the design of interaction techniques that would support a programmer in understanding interaction code. We devised the design principles to fulfill the requirements. Most design principles and techniques leverage on a textual code editor. Even if some graphical representations can be used to represent data-flow or FSMs, text-based editors are still heavily used as they provide features that are deemed usable by many programmers. Still, we hypothesize that the regular, mostly 1D, textual presentation of causal relationship constructs spread in multiple files does not help programmers to understand the causal chain. We demonstrate the interaction techniques in a concrete way through use cases.

4.1 Requirements

The understanding of interaction code poses specific challenges arising from the multiplicity of causal chains across files that split the description of behaviors. We wanted to design interaction techniques directly available in the edited source code file, saving navigation time across files. Although code lines integration within edited source code is not new, we have not seen such techniques applied to interaction code and causal relationships understanding. Hence, our interaction techniques should support programmers in:

- Understanding the causal dependencies [ReqCausal]
- Backtracking the origin of a causal propagation, overcoming the inconvenience of split code over files [ReqNoSplit]
- Visualizing transitions and states activations dynamically [ReqDynamicCaus]

4.2 Design principles

The first design principle that we followed to fulfill [ReqCausal] is to enable the programmer to make the y-ordering of lines of code consistent with the expected execution ordering [DgnYRearrange]. Reordering is especially important when the programmer wants to apprehend the multiplicity of execution paths due to uncontrolled sequences of external events. It is important to note that reordering only concerns the appearance of the program, not its actual source code, and it does not change its semantics. The representations of code change upon users' request, and can be set back to the original arrangement upon request.

The second design principle we followed to fulfill [ReqNoSplit] is to bring together the causal relationships that are far away from each other in the source code, including in the same file [DgnTogether]. This is especially important when coping with causal relationships that reside in several code locations.

The third design principle is to take advantage of the text representation and of the properties of visual variables from Semiology of Graphics [5] to understand the code and its execution. Notably, the first three interactions reorder the programming constructs to display them like an imperative, y-ordered control-flow. Note that the first design principle can be considered a special case of the third one, but its importance deserves a proper principle [DgnVisVar].

Finally, the fourth design principle consists in using animations to help the user apprehend the changes of the representations [DgnAnim] [16, 60], especially for [ReqDynamicCaus]. We also took into consideration the guidelines related to animation of lists, as lines of codes can be considered as lists [60].

To the best of our knowledge, [DgnYRearrange] and [DgnTogether] have never been identified and used in past work. [DgnAnim] is not new, but has not been applied to code-to-code transformation. [DgnVisVar] is not new [17], but has never been applied to interaction code. Though following the same design principles, the interaction techniques were not designed to be completely consistent: our goal was to explore a design space and how well the techniques would support program comprehension. Currently, the interactions are implemented in a GUI that enables a programmer to launch a SMALA program and explore its source code with the interactions. Even though the first three interactions may be available statically (e.g. without running the explored source code), our implementation relies on the run-time *initialization* phase of the tree of processes and the reflexive capabilities of the SMALA execution engine. Only the fourth interaction relies on the run-time *execution*. Relying on the execution engine enabled us to prototype the interactions without implementing a static analyzer.

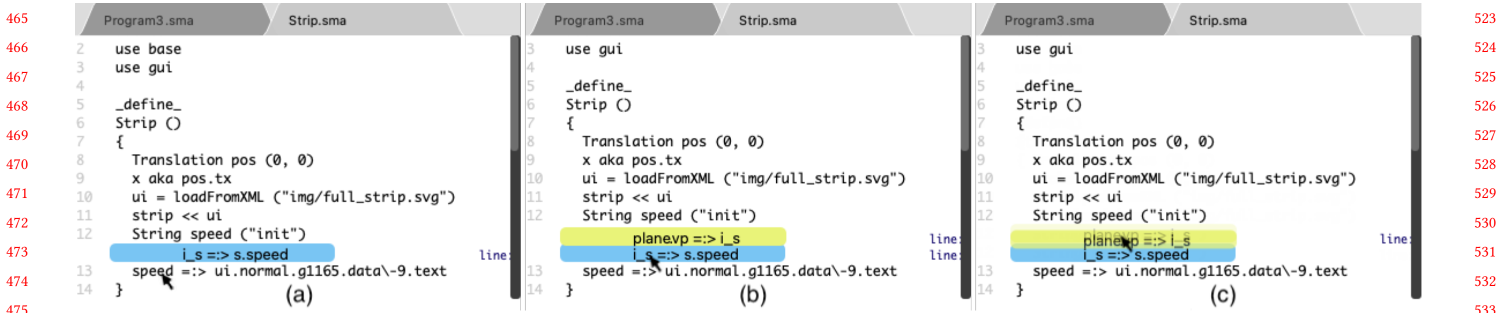

Figure 2: *Interaction 1.* Reordering a data-flow: (a) the user clicks on the left of a connector, an animation inserts the upstream data-flow construct (b) upon click on the new line, an animation inserts the next upstream construct (c) the user clicks on the new line, the system animates the line up-and-down to specify that the beginning of the data-flow has been reached.

4.3 Interaction 1: reordering a data-flow

The first interaction technique provides the programmer with means to navigate inside a data-flow and display it like an imperative control-flow. A use case is as follows. The programmer of a drone ground station application is faced with an unexpected behavior while testing interaction code: some text in the graphical interface is not updated as it should be, which is likely the symptom of a broken data-flow. The cause to be identified is a missing connector: the speed of the drone model displayed in the view is not connected to the corresponding data from the bus.

We designed an interaction technique to navigate the data-flow (Figure 2): clicking on a variable on the left-hand side of a data-flow construct summons an animated apparition of the upstream construct that is connected to the said variable. The upstream construct appears in a line of code just above the clicked construct. In a recursive manner, the programmer can click on the summoned lines to display further upstream constructs. If multiple sources are connected on a property, they are all shown using a line each. If there is nothing connected, a short animation quickly bounces the clicked line up and down. Note that the summoned lines may come from the current text file being edited but also from other files. In this case, the source filename is appended to the right of the line as a hyperlink that enables the programmer to jump to the code.

Step-by-step, the programmer is thus able to trace back the absence of a causal construct that should have led to the activation of the line of interest. In our example, we can see that there is nothing connected to the property named *plane.vp*. As such, this cannot be automatically identified as a mistake, since such behavior could have been perfectly legitimate in some applications (in a program, many properties are partially connected). The interaction also works with downstream constructs by clicking on the right side of the arrow, or on the declaration of a property. It is also available for transitions in FSMs: one can explore the chain that leads to or depart from a transition event.

Interaction 1 relies on three design principles to help understand the data-flow: [DgnTogether], [DgnYArrange] and [DgnVisVar]. Compared to a "Jump to Reference" command in a traditional code editor which completely changes the content of the window, or at best provides a transient pop-up on top of the currently edited window, the insertion above the lines being read keeps the context in which the programmer tries to understand the code, and displays the lines of code as if they were next to each other. The resulting sequence of lines of code, perceptually ordered in the y-dimension and x-aligned [17], is reminiscent of the sequence of lines of code in an imperative language. However, here the y-dimension of the source code is mapped to the causal relationships (reactive language) instead of the program counter (imperative language), while the x-alignment specifies that all lines belong to a same data-flow (reactive) instead of control-flow (imperative). This makes the causal chain directly visible. The same interaction technique makes it easier to spot a wrong link in the data-flow: the programmer can quickly identify a process wrongly connected to another one.

4.4 Interaction 2: reordering *textual* FSMs

In the example in Figure 1, the FSM has 7 states and 20 transitions. It is thus difficult to apprehend the whole FSM code and understand its behavior given a particular sequence of events. The use-case is as follows: there is a suspicious, transient activation of the check-LoopState state. The user wants to understand the causal events that lead to this state, what the state is activating in turn, and what causes its exiting. However, the current graphical state of the FSM representation makes it difficult to visualize such a sequence as it forces the user to look for the involved transitions and to hop from a line to another in arbitrary directions.

We designed an interaction technique based on the reordering of the elements of an FSM. The programmer can hover over a state, which highlights with a green background the transitions that go into or leave the state. S/he can then click on the state and see a smooth, animated change of the layout of the transitions around the clicked state, to make 'in' transitions lie above the state, and 'out' transitions lie below the state. She can continue exploring the follow-up causal chains by clicking on another state, and see the 'in' and 'out' transitions move around it.

Interaction 2 relies on the same design principles as Interaction 1, but applied to the causal chain related to FSMs (with states, transitions, events) in lieu of data-flows (with variables and operators such as connectors, bindings or assignments). Similarly, the resulting sequence of lines of code, ordered in the y-dimension and x-aligned, is reminiscent of the sequence of lines of code in an

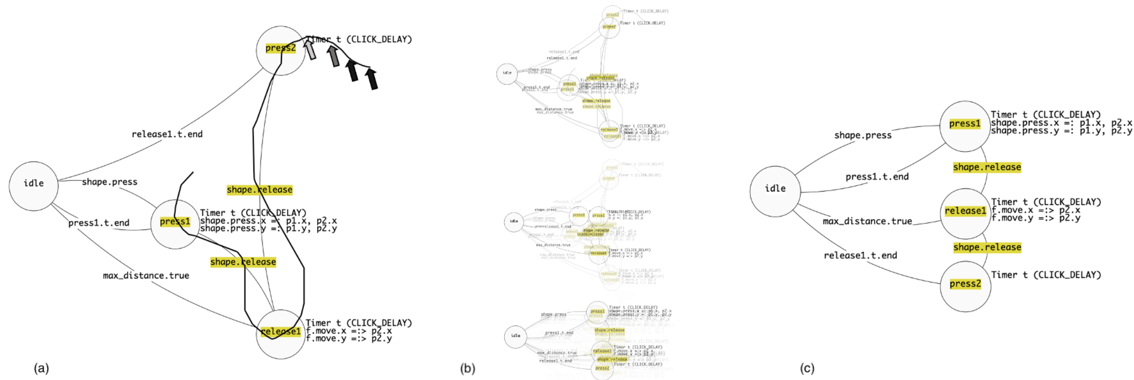

Figure 3: *Interaction 3.* Reordering an FSM given a path selection: (a) starting from a canonical graphical circle-arrow representation of an FSM, the user draws a path through states and transitions according to the causal chain she wants to explore; (b) the system reorders and animates the circles and arrows; (c) the final layout is similar to the corresponding text representation.

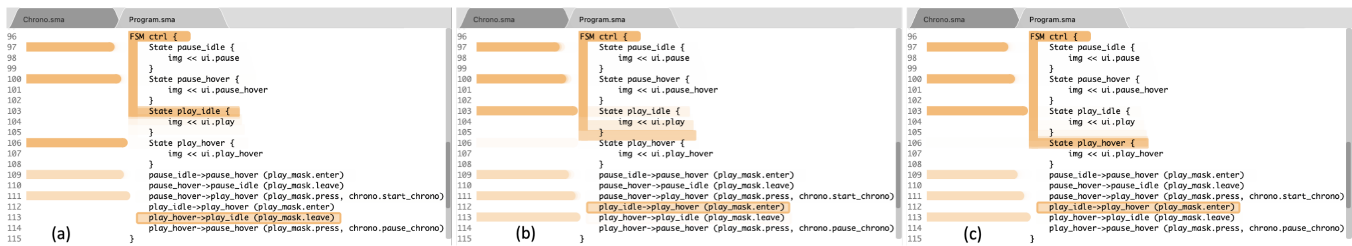

Figure 4: *Interaction 4.* Showing in the text editor the activation of states and transitions during execution. (a) The FSM enters state *play idle*, due to activation of the highlighted transition (b) The selected state moves to *play hover* (c) State *play hover* is highlighted and changes in the history sidebars are updated.

imperative language, and allows a developer to better understand the control-flow and its associated causal chain.

Navigation within FSMs and dataflows can be combined. One can rearrange transitions around a state (interaction 2), and click on the process `to_check_loop` that triggers a transition… (interaction 1):

```
check_state -> checkLoopState(to_check_loop)
```

… to summon the apparition of an upstream binding:

```
checkSound.t.end -> to_check_loop
check_state-> checkLoopState(to_check_loop)
```

4.5 Interaction 3: reordering *graphical* FSMs

The previous interaction techniques are a demonstration of how text-based representations of interaction code could be re-arranged to better follow the control-flow. The use-case from Interaction 2 was concerned with the understanding of the control-flow "around" one particular state. Here the use-case is extended to multiple states.

A popular representation of FSMs relies on circles depicting states and on arrows depicting transitions, annotated with the event that fires a transition. Even if the two representations ("textual code" and "circle-arrow") seem different, we can smoothly transition from one to another to adapt the view according to the task at hand.

We designed an interaction technique that rearranges the circle-arrow representation (Figure 3). Starting from the circle-and-arrow representation, the user draws a line that passes through states, transitions and events. The users should specify their gesture according to the particular sequence they want to analyze. After the gesture has been performed, the system animates a rearrangement of the circle-arrow representation. The final arrangement allows the programmer to read from the top to the bottom the causal chain involving a succession of states and events.

Again, such a representation provides the programmer with a sequential reading of the code, which makes causal ordering salient. Remarkably, the final circle-arrow representation is similar to the textual one. This supports the hypothesis that graphical and textual are not so different [17]): the visual representation reuses the assets of the textual one, such as selectivity of the x-dimension (i.e. left alignment [17]) and the ordered perception of the y-dimension to depict the order of the control-flow.

4.6 Interaction 4: showing the dynamics of FSMs

The dynamic behavior of FSMs can be hard to understand. The use case is a follows: on a GUI there is a displayed clock, coded with a FSM, which should switch from clock to timer mode upon request on a "play" button. But nothing happens. The programmer wants to check whether the issue is a faulty transition. We designed an animation that highlights the activation of states and transitions during execution (see Figure 4). Each time an FSM enters some state, the state is highlighted and the transition that activated it is both

highlighted and outlined. States might be declared remotely from the declaration of the FSM itself, especially in an embedded FSM. Therefore, the representation also connects the activated state (e.g., *play_hover*) to the parent FSM (e.g.,*ctrl*) in which it is declared. As time goes by during execution, the previous activated states and transitions are denoted by horizontal sidebars that progressively fade away and shrink towards the left (see Figure 4), much like a vertical VU-meter in music players. The progressive fading gives the user a sense of the history of activations. This representation also shows whether a state or a transition has been activated at least once. This is particularly useful when the transition between states is very fast.

5 EVALUATION

We conducted a study to assess how much more usable would Causette be compared to a traditional text editor. Due to the nature of our research (facilitating some program understanding tasks), we expected that it would be difficult to design a controlled experiment that would be both statistically and practically significant. We expected that measuring the time of completion of a program understanding task would be highly dependent on inter-individual differences. We thus followed the principles of Single-Subject Research [45]. Single-Subject Research involves testing a small number of participants and focusing intensively on the behavior of each individual and measuring strong and consistent effects that have biological or social importance [45]. In the following, we present the results per-subject, instead of aggregated measures of multiple subjects. To design the experiment and report on its results, we followed the principles of Fair Statistical Communication in HCI [20]. In particular, we asked quantitative research questions when suitable, and we stated the effect size.

5.1 Research questions

We focused on the following research questions:
RQ1. How much would Causette facilitate the understanding of causal structures in interaction code?
RQ2. How much would Causette make a difference when given a complex "interactive bug"?
RQ3. How much would programmers benefit from insertion of related code lines into the source code being analyzed compared to common methods in text editors (e.g. '"find", "jump to definition")?

5.2 Participants

We recruited 10 participants. All of them were men (more on this in section 6) and were of age 20-25, 1 of age 25-30, 4 of age 30-40. They had at least an M.S in Computer Science or HCI, 3 hold a Ph.D degree. Half of them had more than three years of professional experience in interaction programming with Qt, Javascript or Swing. All of them were proficient in SMALA and were using Sublime Text as an IDE. Sublime Text is a typical textual code editor, with features such as those mentioned in RQ3.

5.3 Experimental design

We are aware of the challenges arising in the design of code comprehension tasks and existing work has helped us analyze the limits of our own evaluation study [24]. Our experimental design is inspired

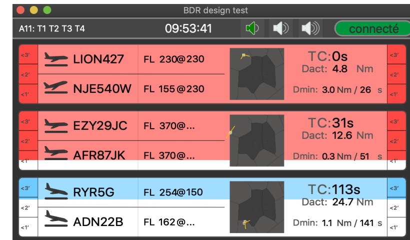

Figure 5: Air Traffic Control app used for the evaluation

by two papers that evaluate new features for IDEs, Theseus [38] and CodeBubbles [8]. We thus conceived a set of tasks related to code comprehension and debugging. We used the code of a real-size application actually in use in Air Traffic Control (ATC). It consists of a GUI that supports air traffic controllers in detecting a conflict between two flights. The source code amounts to approximately 1,000 lines of SMALA spread across 7 files.

Participants were given 6 tasks to complete in total (A to F), those 6 tasks falling into 3 categories (dataflow, value, debug). We designed the tasks within a category to be equivalent in terms of difficulty.

- **Dataflow comprehension questions (A-B):** 2 tasks targeted the understanding of dataflow and required the participants to identify what could trigger the activation of a variable or what the variable could activate down the causal chain.
- **Value questions (C-D):** 2 tasks asked what values a variable could take.
- **Complex comprehension and debugging tasks (E-F):** Finally, 2 tasks involved a complex bug to fix. The tasks require a deeper understanding of the overall behavior of the program.
  E. Missing textual object: The task involved solving a dataflow problem, where the value of the flight callsign was not correctly connected to the textual property of the GUI. The error was a property naming problem due to a former incorrect *copy/paste*.
  F. Faulty alarm: The participant was asked to debug a FSM in charge of the sound management of the application. That FSM had an embedded FSM, with 20 transitions and 9 states. We introduced an error in the code, by deleting one transition of the FSM. As a consequence, the FSM ended up stuck in some state *RadarLoopState* and an alarm kept ringing. Participants had to figure that out and fix it.

There were 2 conditions: a control condition with the participants' usual editor (here Sublime), and a condition with Causette. Sublime is not a full-blown IDE, and we discuss the reasons of choosing it for control condition in section 6. To facilitate within-subjects comparison, each participant was assigned three (from each category) tasks in the control condition and three tasks using Causette. To counterbalance an order effect, half of the participants completed all of their control tasks first, while the other half completed all of their Causette tasks first. The selection of the tasks across categories was randomized.

Before the evaluation, each participant was given a 10 minutes summary of the code: an overview of the architecture and the functions of the app, and the ATC vocabulary to understand identifiers in the code. The code itself did not include any comments. Once

the participant had read the question and was ready to perform the task, a timer was launched. All the interaction techniques were available for each task. Up to seven minutes were allowed for each of the A-B and C-D tasks, while fifteen minutes were allowed for E-F tasks. When participants felt they completed the tasks, they told us, we stopped the task, recorded the duration regardless of the correctness of the answer, and asked for a confidence rate. Participants were also able to give up a task if they felt they could not perform it. If the timer reached the maximum allowed time, we interrupted the task. In total, up to one hour was devoted to the tasks. At the end of the experiment, we made the participants fill a System Usability Scale (SUS) [11] questionnaire on Causette, followed by a 30 minutes interview to get qualitative feedback.

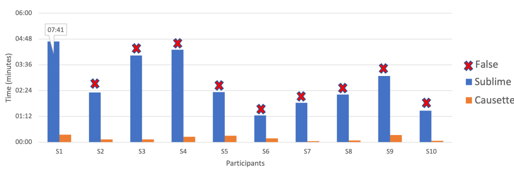

Figure 6: Completion time for the first category of tasks (dataflow questions), comparing control condition (Sublime) and Causette condition. By default, the absence of a red symbol means the answer is correct.

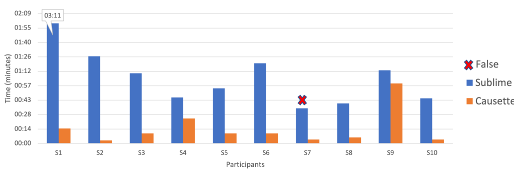

Figure 7: Completion time for the second category of tasks (value questions), comparing control condition (Sublime) and Causette condition.

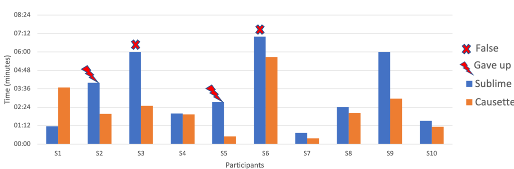

Figure 8: Completion time for the third category of tasks (debugging problems), comparing control condition (Sublime) and Causette condition.

5.4 Results

The complete results of the study [2] include completion time, completion success, confidence rate and SUS score. Figures 6, 7, 8 shows the completion time and completion success per participant for each type of tasks and according to the two conditions.

[2] All data anonymously available here: https://doi.org/10.5281/zenodo.6396517

5.4.1 *Effectiveness - Degree of achievement.*

The participants were all able to correctly complete the tasks in the Causette condition. This contrasts with the completion rate in the Control condition, where many participants could not provide a correct answer or just gave up. In particular, 9 participants out of 10 performed partially correctly the dataflow tasks, as their answers were not exhaustive (they did not mention all paths of the dataflow, and remarkably all but S8 felt confident). Except S1, each participant at least made an error or gave up in the Control condition. 4 participants could not complete the debugging tasks (2 gave incorrect answers, and 2 gave up). This suggests that users are not completely effective with a traditional code editor for these tasks, supporting our analysis of the problems encountered by programmers. This also suggests that our interaction techniques make users more effective than a traditional code editor for all types of task.

Participants were always confident in their answers in the Causette condition. 4 of them (S3, S5, S6, S8) were not confident in some of their answers in the Control condition.

5.4.2 *Efficiency - Time of completion.*

Figure 9 represents the time percentage gain per participant and task groups. Except S1 and S4 in debugging tasks, all participants perform their tasks faster in Causette conditions.

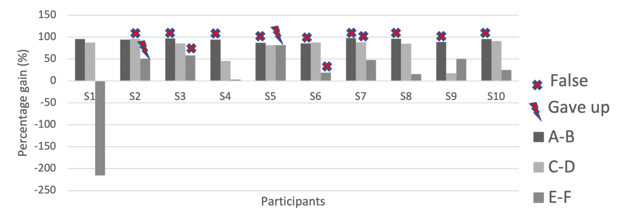

Figure 9: Time percentage gain per participant and per task groups when using Causette compared to Control (Sublime).

In the dataflow tasks, users were able to perform 93% faster with Causette on average. In these tasks, users were only able to answer partially, but we think that the time of completion is meaningful, since they felt they have completed the tasks. Had they resumed the tasks to find out all paths in the dataflow, they would have spent even more time.

In the value tasks, users were able to perform 77% faster with Causette on average.

The results of the debugging tasks are more contrasted. As a reminder, in the control condition, S1 performed better, S4 performed equally, but two participants gave up and two gave incorrect answers. S1 is slower with Causette in the debugging tasks. S1 told us he was familiar with dataflow issues, and less familiar with FSMs bugs, which could explain this outlier. S4 is as fast with Sublime as with Causette. S4 told us he had been focusing on FSMs in his code for two months before the experiment, which could explain his proficiency with Sublime at finding bugs. Except for S1 and S4, the 8 other participants performed faster with Causette. Ignoring incorrect answers, only five participants (S2, S3, S5 S7 and S9) performed significantly faster with Causette with respect to effect size. S6, S8 and S10 performed faster with Causette but with a smaller effect size.

All in all, these results tend to suggest that Causette makes programmers faster at fulfilling the dataflow and value tasks. It may make them faster at debugging with a smaller time gain for half of the participants.

5.4.3 Satisfaction. We gathered the SUS scores for Causette only. 7 out of 10 are higher than 90, the remaining 3 (S1, S6, S7) range from 62,5 to 72,5. S1, S6 and S7 thought that they may need the support of a technical person to use Causette (score 3). S7 eventually stated that he would be able to learn to use it quickly. S1 and S6 did not feel very confident (score 3). S1 and S7 thought the system is not well integrated (score 3). S6 did not think he would use it frequently. We present Causette in this paper as a set of interactions, and not as a system. If we polished the interactions as much as we could, the system itself that embeds them is still not very usable. We hypothesize that the comments of S1, S6 and S7 reflect the usability of the system more than the interactions per se.

5.4.4 Internal validity assessment. We examined whether the design of our experiment exhibits any bias with respect to task equivalence or order (see Figures 10, 11).

In the two conditions, the completion time means between A-B, C-D and E-F tasks tend to confirm that the two tasks within each three categories were of the same difficulty.

In the Causette condition, the order seems to have no effect. There could be an order effect on the completion time of the Sublime condition. It looks as if the use of Causette in a first place had a positive impact on the performances in the Control conditions for the data-flow and value tasks, while it is the opposite for the debugging tasks. However, the computed average times involve at most 3 values, even less when participants gave up or gave an incorrect answer. It is thus difficult to be conclusive or to comment.

Figure 10: Condition ordering effect: completion time means (upper figure) and answer incorrectness (lower figure) per task in Sublime condition.

Figure 11: Condition ordering effect and completion time means per task in Causette condition.

5.4.5 Qualitative results. After each session with a participant, we conducted an interview to get qualitative feedback. Our three research questions RQ1, RQ2, RQ3 target all of the tasks.

RQ1. How much would Causette facilitate the understanding of causal structures in interaction code?

Participants mentioned that they felt Causette saved them time to understand the dataflow (all the participants, except S6). The reordering of the in- and out- transitions was also described as time saving (S1, S4, S8, S10). S1 mentioned that he usually needs to draw FSMs when he got confused about the transitioning between states. Participants S1, S8, S10 pointed out that it was interesting to display the current state of an FSM (in the real-time animation of the FSM) instead of printing it in the console.

The issue about the declarative style of interaction code and how it prevents easy causal understanding was brought up: "Because what's difficult is the declarative aspect, without logical links. You can write it in any order you want." (S8). S8 also developed an issue for FSMs: "with nested FSMs, after a moment I'm lost in the file, and having something that allows to put logical order between states and transitions, that helps."

A few participants mentioned that the tool helped them construct or get quick access to a "mental picture" of the causal relationships: "I don't have to have a mind map all the time [...], there is no need to make drawings." (S4). Another finding was that 6 out of 10 participants underlined what they felt would be the main use of the tool: to get into someone else's code or one's own former project.

The animated insertions were described as presenting the causal relationships effectively. For the dataflow, participants mentioned they felt reassured being provided with a guaranteed, exhaustive list of sources and destinations (S1, S10, S3, S8, S4). S1 and S10 commented on the fact that with a traditional editor, they usually feel unsure whether they figured out completely the dataflow, and they took time to check that the property under study is not dependent on another file or is renamed somewhere else. Such comments are consistent with the confidence rates we measured.

RQ2. How much would Causette make a difference when given a complex "interactive bug"?

The two complex comprehension and debugging tasks were commented by some participants as really reminiscent of the problems they usually face. Several comparisons and equivalent use cases were provided. S3 said "it reminds me of many cases [...] : I was creating my graphical objects, but they were aligned at the top left of my interface, and it took me time to understand that I had not initialized their length and height".

In the debugging task, S5 said he could have answered the last task he gave up in control condition, seeing clearly that the FSM was stuck in a state, but he got tangled up in staying focused on the wrong state. S4 and S7 elaborated on that aspect, saying the FSM animations are useful to locate which state it is blocked in. They found the real time animation and the history bar even more useful "to really see it". S1, S4, S5, S6 found that reordering FSMs make transition errors more salient.

RQ3. How much would programmers benefit from insertion of related code lines into the source code being analyzed?

First, participants brought up that Causette avoids navigating at length: "with the tool we avoid unnecessary navigation. When you use diverse "find" functions, you often end up having too many

occurrences" (S4). This could be supported by the difference in response time we found. Comparisons with search functions in different IDEs were often conducted, to the point where Causette was described as a "super fast search tool" (S5). S2, S3, S4, S5 and S8 liked the simplicity of the triggering of the animated insertion, because it avoided the use of a menu. As one of our participants commented, the interactions involving line insertions might look unusual in the beginning, but it seems easy to get into the habit. Since code is rearranged, it is worth noting users may be disorientated. This was mentioned by one participant: "I clicked unintentionally, and I had trouble finding my way around, I didn't know what I had clicked on" (S9). This is especially true in interactions with FSMs as the layout may end up very different from the initial arrangement. We used animations to mitigate this aspect. We also provided an "original view" button to get the layout back to its original form. As for the FSM, the real-time animation was appreciated but the reordered FSM animation was also found useful. S1 who liked the graphical FSM underlined that the reordered FSM was very interesting.

6 THREATS TO VALIDITY

Participants. The number of participants in the experiment is low, and they are all men. We could not avoid this sample bias, as the users of the Smala language happen to be men. Still, we do not have any plausible hypothesis about the effect of genre on the results of the experiments. As noted by Feitelson [24], the effect of genre is not clear and requires further investigation.

Size of code. The application used during the evaluation is 1,000 lines of code only. One can wonder if the interaction techniques would scale to larger code bases. 1,000 lines seem small, but they are written in a specialized, interaction-oriented programming language which is expressive and should not be compared to the size of code written in a general-purpose language such as Java. Still, the FSM of the example has a moderate complex behavior. Scaling up the number of states to a dozen would make it more difficult to perform interaction 3. Besides, our representations have limits: for example, Interaction 1 and 2 only display one causal chain at a time, and do not allow exploring multiple chains simultaneously. But the control condition does not facilitate exploring one causal chain only neither, and we think that it would scale even more badly, as one would have to navigate more files with longer causal chains.

Tasks. The tasks only partially represent the programming activity. We are aware that the evaluation has limits and should be complemented with other experiments. In particular, tasks A-B and C-D where the results were the most in favor of Causette, may seem idiosyncratic. However, the tasks represent a typical problem encountered by programmers of interactive programs. We designed the interactions to support solving those problems, hence it is not surprising that the evaluation suggests that Causette offers support. Nevertheless, it had to be demonstrated.

Control condition. The choice of a limited set of Sublime Text interactions as the control condition is disputable. Causette is a set of interaction techniques that may have been compared to those of a full-featured IDE. Still, we left the participants choose the Sublime Text interactions they use in their real-world activities. In particular, one can debug program with Sublime Text. However, such a debugger is of little help for the interaction code problems.

Understanding. A general pitfall of comprehension evaluation [24] applies to our work: does the study actually tell something about *comprehension*? It could turn out that some participants, especially for task A-B and C-D, found the *correct answers* about the causal chains in code, without properly *understanding* them. Addressing that issue would require further investigation.

Reading/Writing. Finally, the study suggests that Causette support code reading, but not code writing. Since developers read 10 times more code than they write [24, 41], our results suggest that at least Causette could be useful. Further work is needed to assess whether Causette could support code writing.

Generalizability. The interactions may be too specialized to the particular language we used. Some languages or toolkits provide some of Smala's features e.g. Qt's signal/slot, JavaFX binding, SwingState's FSMs. We think that most of the interactions could be applied to these features e.g. navigating the chain of signal/slots in Qt or SwingState's FSMs. However, this necessitates appropriate analysis and introspection tools. For example, one could use the Qt's MetaObject system to gather dependency information, and use the API provided by some text editors to insert the upstream and downstream signal/slots where a particular connection is created in the code. Similarly, the QML or SwingStates run-times could record the parameters of the transitions and inform Causette. With this, it would be possible to adapt Causette interactions to an editor of Qt code: clicking on $object1.signal$ in this line of code…:

```
object1.signal.connect(object2.slot)
```

…could summon the apparition of an upstream construct above:

```
object0.signal.connect(object1.slot)
object1.signal.connect(object2.slot)
```

Similarly, since QML and Swingstates provide explicit FSMs syntaxes, an editor of such languages could rearrange the order of transitions as in interaction 2.

7 CONCLUSION AND FUTURE WORK

We presented *Causette*, a set of four interaction techniques for a textual and graphical code editor. The interactions rearrange and animate textual code in a way that makes the causal relationships more understandable. An evaluation with professional programmers suggests that Causette may be more usable than a regular text editor for some interaction programming tasks. This work indicates that rearranging interaction code may help developers better understand and fix it. This work can be continued by exploring unanswered questions on scalability, generalization, disorientation, and ecological validity. Instead of running another controlled experiment, we plan to provide our participants with a better, more robust, and more integrated version of our tool and observe its use in practice during a longitudinal study. This should also provide us with new insights on how to best support interaction programmers.

ACKNOWLEDGMENTS

We thank our participants and colleagues for their advices and comments. This work was partly supported by the French « Programme d'Investissements d'avenir » ANR-17-EURE-0005 conducted by ANR and by Agence de l'Innovation de Défense.

REFERENCES

[1] Caroline Appert and Michel Beaudouin-Lafon. 2006. SwingStates: Adding State Machines to the Swing Toolkit. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology* (Montreux, Switzerland) *(UIST '06)*. Association for Computing Machinery, New York, NY, USA, 319–322. https://doi.org/10.1145/1166253.1166302

[2] Caroline Appert and Michel Beaudouin-Lafon. 2008. SwingStates: Adding state machines to the swing toolkit. In *UIST 2006: Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*. https://doi.org/10.1145/1166253.1166302

[3] Gérard Berry. 2000. The Foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, C. Stirling G. Plotkin and M. Tofte (Eds.). MIT Press.

[4] Gérard Berry and Manuel Serrano. 2020. HipHop.Js: (A)Synchronous Reactive Web Programming. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 533–545. https://doi.org/10.1145/3385412.3385984

[5] Jacques Bertin. 1983. *Semiology of Graphics*. University of Wisconsin Press.

[6] Alan Blackwell and Thomas Green. 2003. Chapter 5 - Notational Systems - The Cognitive Dimensions of Notations Framework. In *HCI Models, Theories, and Frameworks*, John M. Carroll (Ed.). Morgan Kaufmann, San Francisco, 103–133. https://doi.org/10.1016/B978-155860808-5/50005-8

[7] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010. Code bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings - International Conference on Software Engineering*. https://doi.org/10.1145/1806799.1806866

[8] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. Laviola. 2010. Code bubbles: A working set-based interface for code understanding and maintenance. In *Conference on Human Factors in Computing Systems - Proceedings*. https://doi.org/10.1145/1753326.1753706

[9] Alexander Breckel and Matthias Tichy. 2016. Embedding programming context into source code. In *IEEE International Conference on Program Comprehension*. https://doi.org/10.1109/ICPC.2016.7503732

[10] Victor Bret. 2012. Learnable Programming–Designing a programming system for understanding programs. http://worrydream.com/LearnableProgramming. http://worrydream.com/LearnableProgramming Accessed: 2021-08-20.

[11] John Brooke. 1986. SUS: A 'Quick and Dirty' Usability Scale. In *Usability Evaluation In Industry*, McClelland A.L. Jordan P.W., Thomas B, Weerdmeester B.A. (Ed.). Taylor and Francis, London.

[12] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye Movements in Code Reading: Relaxing the Linear Order. In *IEEE International Conference on Program Comprehension*. https://doi.org/10.1109/ICPC.2015.36

[13] Stéphane Chatty. 1994. Extending a Graphical Toolkit for Two-Handed Interaction. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology* (Marina del Rey, California, USA) *(UIST '94)*. Association for Computing Machinery, New York, NY, USA, 195–204. https://doi.org/10.1145/192426.192500

[14] Stéphane Chatty. 2008. Programs = data + algorithms + architecture: Consequences for interactive software engineering. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-540-92698-6_22

[15] Fanny Chevalier, Pierre Dragicevic, Anastasia Bezerianos, and Jean Daniel Fekete. 2010. Using text animated transitions to support navigation in document histories. In *Conference on Human Factors in Computing Systems - Proceedings*. https://doi.org/10.1145/1753326.1753427

[16] Fanny Chevalier, Nathalie Henry Riche, Catherine Plaisant, Amira Chalbi, and Christophe Hurter. 2016. Animations 25 Years Later: New Roles and Opportunities. In *Proceedings of the International Working Conference on Advanced Visual Interfaces, AVI 2016, Bari, Italy, June 7-10, 2016*, Paolo Buono, Rosa Lanzilotti, Maristella Matera, and Maria Francesca Costabile (Eds.). ACM, 280–287. https://doi.org/10.1145/2909132.2909255

[17] Stéphane Conversy. 2014. Unifying textual and visual: A theoretical account of the visual perception of programming languages. In *Onward! 2014 - Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Part of SPLASH 2014*. https://doi.org/10.1145/2661136.2661138

[18] Michael Desmond, Margaret Anne Storey, and Chris Exton. 2006. Fluid source code views. In *IEEE International Conference on Program Comprehension*. https://doi.org/10.1109/ICPC.2006.24

[19] Martin Dias, Diego Orellana, Santiago Vidal, Leonel Merino, and Alexandre Bergel. 2020. Evaluating a visual approach for understanding javascript source code. In *IEEE International Conference on Program Comprehension*. https://doi.org/10.1145/3387904.3389275

[20] Pierre Dragicevic. 2016. Fair Statistical Communication in HCI. In *Modern Statistical Methods for HCI*. Springer, 291 – 330. https://doi.org/10.1007/978-3-319-26633-6_13

[21] Pierre Dragicevic, Stéphane Huot, and Fanny Chevalier. 2011. Gliimpse: Animating from markup code to rendered documents and vice versa. In *UIST'11 - Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. https://doi.org/10.1145/2047196.2047229

[22] Niklas Elmqvist and Philippas Tsigas. 2003. Causality visualization using animated Growing Polygons. In *Proceedings - IEEE Symposium on Information Visualization, INFO VIS*. https://doi.org/10.1109/INFVIS.2003.1249025

[23] Niklas Elmqvist and Philippas Tsigas. 2003. Growing Squares: Animated Visualization of Causal Relations. In *Proceedings of ACM Symposium on Software Visualization*.

[24] Dror G. Feitelson. 2021. Considerations and Pitfalls in Controlled Experiments on Code Comprehension. In *IEEE International Conference on Program Comprehension*. https://doi.org/10.1109/ICPC52881.2021.00019 arXiv:2103.08769

[25] 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* (2014). https://doi.org/10.1109/TVCG.2014.2346435

[26] T.R.G. Green and M. Petre. 1992. When Visual Programs are Harder to Read than Textual Programs. *Human-Computer Interaction: Tasks and Organisation* (1992).

[27] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The Synchronous Data Flow Programming Language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

[28] R. Jacob. 1996. A visual language for non-WIMP user interfaces. *Proceedings 1996 IEEE Symposium on Visual Languages* (1996), 231–238.

[29] Ahmad Jbara. 2020. SimplyHover: Improving comprehension of else statements. In *IEEE International Conference on Program Comprehension*. https://doi.org/10.1145/3387904.3389297

[30] Ahmad Jbara and Dror G. Feitelson. 2014. On the effect of code regularity on comprehension. In *22nd International Conference on Program Comprehension, ICPC 2014 - Proceedings*. https://doi.org/10.1145/2597008.2597140

[31] Alan C. Kay. 1993. The Early History of Smalltalk. *SIGPLAN Not.* 28, 3 (March 1993), 69–95. https://doi.org/10.1145/155360.155364

[32] Andrew J. Ko and Brad A. Myers. 2004. Designing the Whyline: A debugging interface for asking questions about program behavior. In *Conference on Human Factors in Computing Systems - Proceedings*.

[33] Andrew J. Ko and Brad A. Myers. 2008. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings - International Conference on Software Engineering*. https://doi.org/10.1145/1368088.1368130

[34] 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* (2006). https://doi.org/10.1109/TSE.2006.116

[35] P. J. Landin. 1966. The next 700 Programming Languages. *Commun. ACM* 9, 3 (March 1966), 157–166. https://doi.org/10.1145/365230.365257

[36] Thomas D. LaToza and Brad A. Myers. 2010. Developers ask reachability questions. In *Proceedings - International Conference on Software Engineering*. https://doi.org/10.1145/1806799.1806829

[37] Catherine Letondal, Stéphane Chatty, W Greg Phillips, and Fabien André. 2010. Usability requirements for interaction-oriented development tools. In *PPIG*.

[38] Tom Lieber, Joel Brandt, and Robert C. Miller. 2014. Addressing misconceptions about code with always-on programming visualizations. In *Conference on Human Factors in Computing Systems - Proceedings*. https://doi.org/10.1145/2556288.2557409

[39] Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions about Code with Always-on Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. Association for Computing Machinery, New York, NY, USA, 2481–2490. https://doi.org/10.1145/2556288.2557409

[40] Mathieu Magnaudet, Stephane Chatty, Stephane Conversy, Sebastien Leriche, Celia Picard, and Daniel Prun. 2018. Djnn/smala: A conceptual framework and a language for interaction-oriented programming. *Proceedings of the ACM on Human-Computer Interaction* (2018). https://doi.org/10.1145/3229094

[41] Robert C. Martin. 2014. *Clean Code - A Handbook of Agile Software Craftmanship*. arXiv:arXiv:1011.1669v3

[42] Microsoft. 2013. CodeLens. https://docs.microsoft.com/en-us/visualstudio/ide/find-code-changes-and-other-history-with-codelens?view=vs-2022

[43] Roberto Minelli, Andrea Mocci, Romain Robbes, and Michele Lanza. 2016. Taming the IDE with fine-grained interaction data. In *IEEE International Conference on Program Comprehension*. https://doi.org/10.1109/ICPC.2016.7503714

[44] D. Moody. 2009. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering* 35, 6 (2009), 756–779. https://doi.org/10.1109/TSE.2009.67

[45] David L. Morgan and Robin K. Morgan. 2017. Single-Case Research Methods for the Behavioral and Health Sciences. In *Research Methods in Psychology*. https://

opentext.wsu.edu/carriecuttler/chapter/overview-of-single-subject-research/
[46] Brad A. Myers. 1991. Separating application code from toolkits: Eliminating the Spaghetti of call-backs. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology, UIST 1991*.
[47] Brad A. Myers. 2013. Improving program comprehension by answering questions (keynote). In *IEEE International Conference on Program Comprehension*. https://doi.org/10.1109/ICPC.2013.6613827
[48] Brad A. Myers, Dario Giuse, Andrew Mickish, Brad Vander Zanden, David Kosbie, Richard McDaniel, James Landay, Matthew Goldberg, and Rajan Pathasarathy. 1994. The garnet user interface development environment. In *Conference on Human Factors in Computing Systems - Proceedings*. https://doi.org/10.1145/259963.260472
[49] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, and Bruce D. Kyle. 1997. The amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering* (1997). https://doi.org/10.1109/32.601073
[50] Brad A. Myers, John F. Pane, and Andy Ko. 2004. Natural programming languages and environments. https://doi.org/10.1145/1015864.1015888
[51] Stephen Oney, Brad Myers, and Joel Brandt. 2014. InterState: A language and environment for expressing interface behavior. In *UIST 2014 - Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*. https://doi.org/10.1145/2642918.2647358
[52] Norman Peitek, Janet Siegmund, and Sven Apel. 2020. What drives the reading order of programmers? an eye tracking study. In *IEEE International Conference on Program Comprehension*. https://doi.org/10.1145/3387904.3389279
[53] Martin Pinzger, Katja Gräfenhain, Patrick Knab, and Harald C. Gall. 2008. A tool for visual understanding of source code dependencies. In *IEEE International Conference on Program Comprehension*. https://doi.org/10.1109/ICPC.2008.23
[54] Darrell R Raymond. 1991. Reading source code. *Proc. CASCON* (1991).
[55] Steven P. Reiss, Jared N. Bott, and Joseph J. Laviola. 2012. Code bubbles: A practical working-set programming environment. In *Proceedings - International Conference on Software Engineering*. https://doi.org/10.1109/ICSE.2012.6227235
[56] Steven P. Reiss and Alexander Tarvo. 2013. Tool demonstration: The visualizations of code bubbles. In *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*. https://doi.org/10.1109/VISSOFT.2013.6650521
[57] Wesley C. Salmon. 1984. *Scientific Explanation and the Causal Structure of the World*. Princeton University Press.
[58] Guido Salvaneschi and Mira Mezini. 2016. Debugging for reactive programming. In *Proceedings - International Conference on Software Engineering*. https://doi.org/10.1145/2884781.2884815
[59] Guido Salvaneschi and Mira Mezini. 2016. Debugging reactive programming with reactive inspector. In *Proceedings - International Conference on Software Engineering*. https://doi.org/10.1145/2889160.2893174
[60] Céline Schlienger, Stéphane Conversy, Stéphane Chatty, Magali Anquetil, and Christophe Mertz. 2007. Improving Users' Comprehension of Changes with Animation and Sound: An Empirical Assessment, Vol. 4662. 207–220. https://doi.org/10.1007/978-3-540-74796-3_20
[61] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2006. Questions programmers ask during software evolution tasks. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. https://doi.org/10.1145/1181775.1181779
[62] Dong Bach Vo, Kristina Lazarova, Helen C. Purchase, and Mark McCann. 2020. Visual Causality: Investigating Graph Layouts for Understanding Causal Processes. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-030-54249-8_26
[63] Anneliese Von Mayrhauser and A. Marie Vans. 1997. Program understanding behavior during debugging of large scale software. In *Papers Presented at the 7th Workshop on Empirical Studies of Programmers, ESP 1997*. https://doi.org/10.1145/266399.266414
[64] Zhanyong Wan and Paul Hudak. 2000. Functional Reactive Programming from First Principles. *SIGPLAN Not.* 35, 5 (May 2000), 242–252. https://doi.org/10.1145/358438.349331