Design As Exploration: Creating Interface Alternatives through Parallel Authoring and Runtime Tuning

Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, Scott R. Klemmer
Stanford University HCI Group
Computer Science Department, Stanford, CA 94305
{bjoern, lorenyu, aallison, yyang1, srk}@cs.stanford.edu

ABSTRACT

Creating multiple prototypes facilitates comparative reasoning, grounds team discussion, and enables situated exploration. However, current interface design tools focus on creating *single* artifacts. This paper introduces the Juxtapose code editor and runtime environment for designing *multiple* alternatives of both application logic and interface parameters. For rapidly comparing code alternatives, Juxtapose introduces selectively parallel source editing and execution. To explore parameter variations, Juxtapose automatically creates control interfaces for "tuning" application variables at runtime. This paper describes techniques to support design exploration for desktop, mobile, and physical interfaces, and situates this work in a larger design space of tools for explorative programming. A summative study of Juxtapose with 18 participants demonstrated that parallel editing and execution are accessible to interaction designers and that designers can leverage these techniques to survey more options, faster.

Author Keywords
Design alternatives, prototyping, design tools.

ACM Classification Keywords
H.5.2. [Information Interfaces]: User Interfaces—*prototyping.* D.2.2 [Software Engineering]: Design Tools and Techniques—*user interfaces*.

INTRODUCTION

"In engineering, enlightened trial and error, not the planning of flawless intellects, has brought most advances; this is why engineers build prototypes." –Eric Drexler [9]

When designers create multiple distinct prototypes prior to committing to a final direction, several important benefits arise. First, alternatives provide designers with a more complete understanding of a design space [12]. Second, developing different "what if" scenarios enables more effective, efficient decision making within organizations [28]. Third, discussing multiple prototypes helps project stakeholders better communicate their requirements [18]. Finally, presenting multiple alternatives in user studies facilitates participants' ability to understand design tradeoffs and offer critical feedback [33].

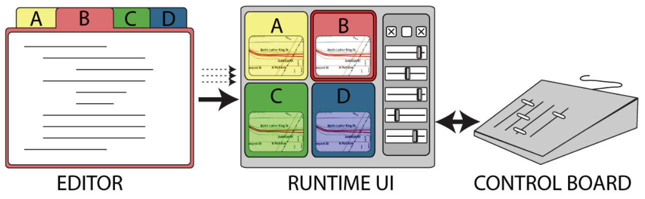
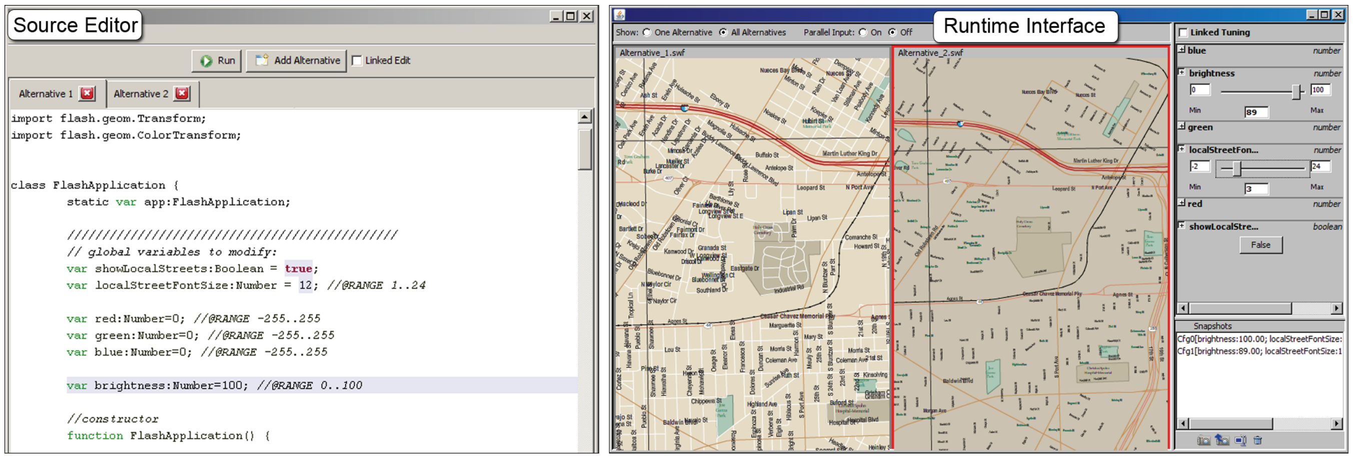


Figure 1. Interaction designers explore options in Juxtapose through a source code editor that supports alternative code documents *(left)*, a runtime interface that offers parallel execution and tuning of application parameters *(center)*, and an external controller for spatially-multiplexed input *(right)*.

Placing "enlightened trial and error" at the core of design raises the research question, *how might authoring environments support designers in creating and managing design options?* Traditionally, design tools have focused on creating *single* artifacts [30]. Research in subjunctive interfaces [21] pioneered techniques for parallel exploration of multiple scenarios during information exploration. Set-based interaction techniques have also been introduced for graphic design [31, 32] and 3D rendering [23]. Providing alternative-aware tools for *interaction design* adds the challenge of working with two distinct representations: source code, where changes are *authored*; and the running program, where changes are *observed*.

This paper suggests that interaction design tools can successfully scaffold exploration by managing alternatives across source and execution environments, and introduces Juxtapose, an authoring tool manifesting this idea (see Figure 1). This paper makes two contributions to design tool research.

First, it introduces a programming environment in which interaction designers *create and run multiple program alternatives* in parallel. Juxtapose extends *linked editing* [34], a technique to selectively modify source duplicates simultaneously, by turning source alternatives into a set of programs that are executed in parallel (see Figure 2, left). The Juxtapose runtime environment enables interacting with these parallel alternatives (see Figure 2, right).



```
import flash.geom.Transform;
import flash.geom.ColorTransform;


class FlashApplication {
        static var app:FlashApplication;

        ///////////////////////////////////////////////
        // global variables to modify:
        var showLocalStreets:Boolean = true;
        var localStreetFontSize:Number = 12; //@RANGE 1..24

        var red:Number=0; //@RANGE -255..255
        var green:Number=0; //@RANGE -255..255
        var blue:Number=0; //@RANGE -255..255

        var brightness:Number=100; //@RANGE 0..100

        //constructor
        function FlashApplication() {
```

```
blue            number
brightness      number
0               100
Min     89      Max
green           number
localStreetFon...  number
-2              24
Min     3       Max
red             number
showLocalStre...   boolean
False
```

```
Cfg0[brightness:100.00; localStreetFontSize:
Cfg1[brightness:89.00; localStreetFontSize:1
```

Figure 2. In the Juxtapose source editor (left) users work with code alternatives in tabs. Users control whether modifications affect all alternatives or just the presently active one through linked editing. In the runtime interface (right) code alternatives are executed in parallel. Designers tune application parameters with automatically generated control widgets.

Second, Juxtapose introduces "tuning" of interface parameters at runtime by *automatically generating a control interface* for application parameters through source code analysis and language reflection. We hypothesize that runtime controls encourage real-time *improvisation* and *exploration* of the application's parameter space. Designers can save parameter settings in *presets* that Juxtapose maintains across alternatives and executions. To facilitate simultaneous control over multiple tuning parameters, a physical, spatially-multiplexed control surface is supported.

This paper first introduces findings from formative interviews that motivate our work. We then describe the key interaction techniques for creating, executing, and modifying alternatives with Juxtapose. We describe implementations for desktop, mobile, and tangible applications. Next, we discuss tradeoffs of our approach and conclude by presenting evaluation results and an outlook to future work.

FORMATIVE INTERVIEWS

To augment intuitions from our own teaching and practice, we conducted three interviews with interaction designers. Here, we briefly summarize the insights gained.

First, arriving at a satisfying user experience requires simultaneous adjustment of multiple interrelated parameters. For example, a museum installation developer shared that getting an interactive simulation to "feel right" required time-intensive experimentation with parameter settings. Similarly, an instructor for a camera-based interaction design course reported that students found adjusting recognition algorithm parameters to be a lengthy trial-and-error process.

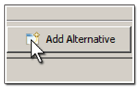
Second, creating alternative program flows is a complementary practice to parameter tuning. In one participant's code, we saw multiple alternative code strategies living side-by-side inside a single function. To try out these different approaches in succession, this interviewee would change which alternative was uncommented (*i.e.*, active), recompile, and execute.

Lastly, all interviewees reported writing custom control interfaces for internal program variables when they were unsure how to find good values. These tuning interfaces are not actually part of the functionality of the application — they function exclusively as exploratory development tools.

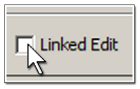
Across the three concerns, interviewees resorted to ad-hoc practices that allowed for some degree of exploration despite a lack of tool support. The following scenario illustrates how Juxtapose can improve such exploration by explicitly addressing parameter variation, alternative creation and control interface generation.

EXPLORING OPTIONS WITH JUXTAPOSE

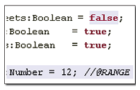
Tina is designing the graphical interface for a new handheld GPS device that both pedestrians and bicyclists will use. She imagines pedestrians will pan the map by tilting the device, and use buttons for zooming. Bicyclists mount the device in a fixed position on their handlebars, so they will need buttons to pan and zoom.



Create new alternative



Toggle parallel editing

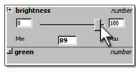


```
:ets:Boolean = false;
:Boolean    = true;
::Boolean   = true;

Number = 12; //@RANGE
```

Highlights show source differences

Alternatives run side by side



Tuning widgets

To try out navigation options, Tina loads her existing map prototype and clicks the *Add Alternative* button; this duplicates her code in a new tab. With the *Linked Edit* box checked, she adds a function to respond to button input. This code change propagates to both alternatives. She clears the *Linked Edit* checkbox so that she can write distinct input handlers in the function body of each alternative. In unlinked mode, edits only apply to the active tab. A colored background highlights code that differs between alternatives.

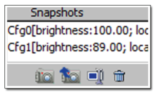
Tina executes her designs. Juxtapose's runtime interface shows the application output of each code alternative side-by-side. One alternative is active, indicated by a red outline. Global `Number` and `Boolean`-typed variables of this alternative are displayed in a variable panel to the right of the running applications. Tina expands the entries for layer visibility, panning speed and zoom step size to reveal *tuning widgets* that allow her to change values of each variable interactively. Tina uses the tuning widgets to arrive at fluid pan and zoom animations.

Sliders for parallel control

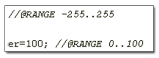
Show:
One Alternative
[✓ All Alternatives]

Show / hide Alternatives

Tina also hypothesizes that bicyclists will value velocity-contingent visual and typographic levels of detail. To adjust the text sizes of multiple road types simultaneously, she moves her non-dominant hand to an external physical control board. She places one finger on each slider, and quickly moves multiple sliders simultaneously to visually understand the gestalt design tradeoffs, such as legibility and clutter. To focus in on the details of one alternative, she toggles between viewing alternatives side-by-side, and viewing just one alternative.


Snapshots
Cfg0[brightness:100.00; loc
Cfg1[brightness:89.00; loca

Snapshots save configurations


```
//@RANGE -255..255

er=100; //@RANGE 0..100
```
Range annotations

Linked Tuning

Toggle parallel tuning

Tina finds several promising parameter combinations for showing levels of detail and uses the *snapshot panel* to save them. Back in the code editor, she introduces a *speed* variable to simulate sensed traveling velocity, and adds code to load different snapshots from the Juxtapose environment when the *speed* variable changes. To constrain tuning to useful values, she adds *range annotation comments*, *e.g.*, indicating that speed should vary between 1 and 30 mph. She runs her design again and selects *speed* for tuning. Moving the associated slider now switches between the snapshot values she previously saved. She checks the *Linked Tuning* box to propagate changes in simulated speed to all alternatives in parallel.

ARCHITECTURE FOR ALTERNATIVE DESIGN

This section outlines fundamental requirements for parallel editing, execution, and tuning, and describes how the Juxtapose implementation supports these techniques.

Parallel Editing

To make working with multiple code alternatives feasible, an authoring environment must keep track of code differences across alternatives, make this structure visually apparent to the user, and offer efficient interaction techniques for manipulating content across alternatives. To support these three requirements, Juxtapose extends Toomim *et al.*'s linked editing technique [34]: alternatives are accessible through document tabs; source differences between tabs are highlighted with a shaded background; and edits can be either local to one alternative or global to all alternatives. Toomim's work focused on sharing code snippets across different locations within a project. Juxtapose instead targets creation of sets of applications based on a core of shared code. To enable interactive editing across multiple documents, Juxtapose replaces Toomim's algorithm with incremental correspondence tracking during editing and

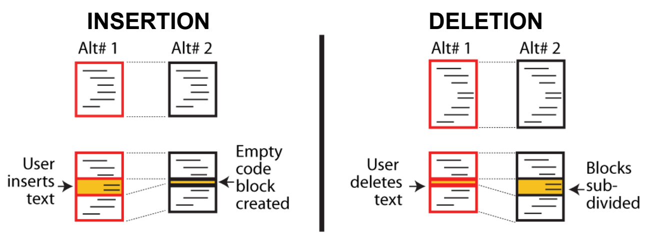


Figure 3. Juxtapose's implementation of linked editing is based on maintaining block correspondences between alternatives across document modifications.

slower content differencing during compilation. The efficiency gains thus realized enable Juxtapose to run comparisons after each key press. Average times for single character-replace operations were under 1 ms with up to 5 alternatives on a 2 GHz PC running Windows Vista.

Juxtapose tracks correspondences between alternatives by partitioning all source alternatives into corresponding blocks. In linked editing, the block *structure* stays fixed and block *content* is modified in all alternatives. In unlinked editing, code blocks are subdivided and alternatives store different content in their sub-blocks (see Figure 3). When inserting text while unlinked, Juxtapose's data structure splits the code into pre- and post-insertion blocks and creates a new code block for the inserted text. Juxtapose splits all alternatives, inserting an empty element into the unmodified alternatives. Deletions also split code blocks. Here, the active document represents the deletion with an empty element; the corresponding elements in the other alternatives contain the deleted text. Code modifications are expressed as deletions followed by insertions. Blocks are never merged during editing.

Incremental structure tracking performs differently than content-based matching if a user types identical code into corresponding locations in two distinct documents: content-based approaches will mark this as a match; structure-based approaches will not. To obtain both interactive performance and content matching, Juxtapose optimizes global block structure with a slower longest-common subsequence algorithm at convenient times (*i.e.*, when compilation is started).

Parallel Execution and Tuning

Executing a set of related interaction designs raises two principal questions: Should alternatives be *presented* in series or in parallel? And should users *interact* with these alternatives one-at-a-time or simultaneously? To investigate how different target devices offer unique opportunities for parallel input and output, we implemented versions of the Juxtapose environment for three domains: *desktop* interactions written in ActionScript for Adobe Flash; *mobile* phone interactions for Flash Lite; and *physical* interactions based on the Arduino microcontroller platform. The three implementations share a common editor but differ in their runtime environment. We discuss each in turn.

Desktop
Desktop PCs offer sufficient screen resolution to run alternative interactions side-by-side, analogous to application windows. Alternatives here are programs authored in ActionScript 2, from which Juxtapose generates a set of Flash movie files using the MTASC compiler [5]. The generated files are then embedded into the Juxtapose Java runtime interface using a Windows-native wrapper library [3]. For consistency with the temporally multiplexed input of windowed operating systems, only one active alternative receives keyboard and mouse input events by default. However, Juxtapose offers the option to replicate user input across alternatives through *event echoing* [20]. By using a provided custom mouse class, mouse events can be intercepted in the active alternative and injected into all other alternatives, which then show a *ghost cursor*. This parallelism only operates at the low level of mouse move and click events, which is useful when both application logic and visual layout are similar across alternatives. However, in absence of a model that translates abstract events in one application into equivalent events in another, users cannot usefully interact with different application logic simultaneously. While development of an abstract input model that provides such a mapping is certainly possible, it is unlikely to occur during prototyping, when the application specification is still largely in flux.

To accomplish runtime variable tuning, bi-directional data exchange between the user's application and the tuning interface is required. On startup, the application transmits variable names, types, and values to Juxtapose. The tuning interface in turn sends value updates for variables to the application whenever its widgets are used. Loading snapshots defined in the tuning interface from code is initiated by a request from the user application, followed by a response from Juxtapose. To accomplish this communication, the user adds a Juxtapose library module to their code. In our implementation, communication between the Flash application and the hosting Java environment takes place through a message-passing protocol and synchronous remote procedure call interface built on top of the Flash Player API.

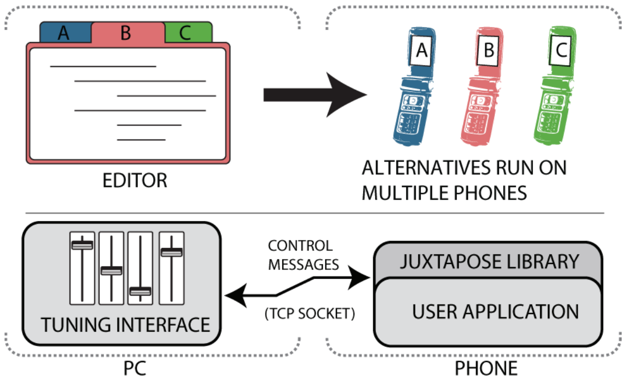


Figure 4. When using Juxtapose Mobile, code alternatives are executed on different phones in parallel. Variable tuning is accomplished through wireless communication between the phone and the Juxtapose tuning interface.

Mobile Phone
For smart phones, the most useful unit of abstraction for parallel execution is not an application window on a handset, but rather the entire handset itself. The small form factor and comparatively lower cost make it attractive to leverage multiple physical devices in parallel (see Figure 4). In Juxtapose mobile, developers still compose and compile applications on a PC. At runtime, the tuning interface resides on the PC, and the alternatives run on different handsets. A designer can thus rapidly switch between alternatives by putting one phone down and picking another one up. To target tuning events to an application running on one particular phone, Juxtapose offers alternative selection buttons in the runtime interface.

Our Juxtapose mobile prototype generates binaries which run on the Flash Lite 2.0 player on Nokia N93 smart phones. The desktop tuning interface and the smart phone communicate through network sockets. When designers run an application on the mobile phone, it opens a persistent TCP socket connection to the Juxtapose runtime interface on the PC. Our prototype uses Wi-Fi for simplicity. Informally, we found that the phone receives variable updates at ~5Hz, much slower than on the PC, but still sufficient for interactive tuning. Response rates are slower because mobile devices trade off increased battery life for slower network throughput and increased latency. A limitation of the current Juxtapose mobile implementation is that users must manually upload compiled files to the phones and launch them within the Flash Lite player. This is due to restrictions of the phone's security architecture.

Physical Interactions
Many interaction designers work with microcontrollers when developing new physical interfaces because of their accessible interface to sensors and actuators. The primary difference to both desktop and mobile development is that novel physical interaction design involves building custom hardware, which is resource intensive. Consequently, designers are likely to embed multiple different opportunities for interaction into the same physical prototype.

Juxtapose supports developing for the Arduino [1] platform and language, a combination popular with interaction designers and artists. Code for all alternatives is cross-compiled with the AVR-GCC compiler suite. Juxtapose for Arduino uploads and runs only one code alternative on one attached Arduino board at a time. When the designer switches between alternatives, Juxtapose transparently replaces the binary running on the microcontroller (see Figure 5) through a bootloader.

Real-time tuning of variables requires a mapping from variable names to types and storage locations, which is not available in the C language that Arduino uses. Juxtapose constructs this map using a preprocessing step that transforms a user's program before compilation. The user's source code is parsed to build a table of global variable

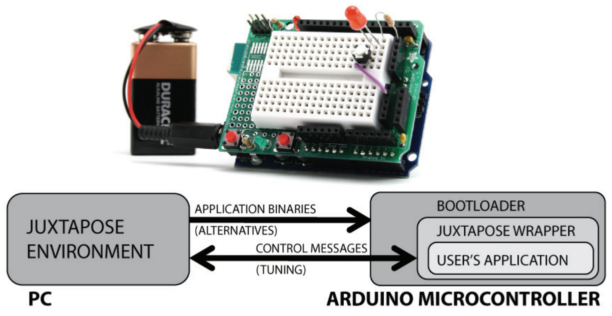


Figure 5. For microcontroller applications, Juxtapose transparently swaps out binary alternatives using a bootloader. Tuning is accomplished through code wrapping.

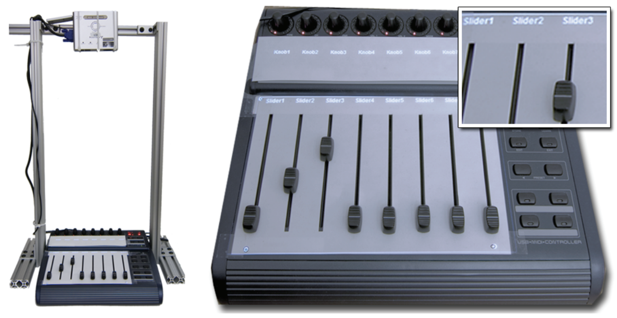


Figure 6. An external controller enables rapid surveying of multidimensional spaces. Variables names are displayed on top of assigned controls to facilitate mapping.

names, types, and pointers to their memory locations. The source is then wrapped in Juxtapose-specific initialization code, into which the variable table is emitted as C code. When a variable is tuned, the embedded wrapper code uses this table to find a pointer to the correct runtime variable from its name and changes the value of the memory location. The wrapper code also contains communication functions to exchange information between microcontroller and PC through a serial port. Some price must be paid for this added flexibility. The developer has to relinquish control of a hardware serial port, and application state is lost whenever alternatives are switched. Snapshots provide a way to save and restore values across such changes.

Writing Tunable Code

Ideally, programmers should be able to leverage tuning and alternatives in their project without changing their source. In practice, tuning is invisible unless modified parameter values have some observable effect on program execution. In other words, the changed variable has to be read again and some action has to be taken based on its value after it was modified at runtime. Thus programmers may have to write additional code that is solely concerned with making their application tunable.

To help programmers express the logic for runtime updates, callback functions provide a lightweight harness: whenever a variable is tuned at runtime, the application is notified of the parameter name and its updated value. In ActionScript, this callback facility is already provided on the language level by the *Object.watch()* method. The following example calls a redraw routine whenever the variable `tunable` is updated by the Juxtapose tuning UI:

```
01  var tunable = 5; //@RANGE 0..100
02  var counter; //@IGNORE
03  var callback= function(varName,oldVal,newVal){
04    redraw();
05    return newVal;
06  }
07  this.watch('tunable',callback);
```

Beyond callbacks, protocols to communicate information *from the source code to the runtime interface* enable designers to initialize the runtime UI programmatically. Programmers can initialize minimum and maximum values for `Number` variables through comment annotations (line 1). They can also hide variables for which tuning is not useful, *e.g.*, counters, from the variable list (line 2). Code annotations have been used in other projects as a source of meta-information, *e.g.*, for labeling different experimental conditions for user testing [22]. Juxtapose currently uses code comments to capture annotations; this functionality could become part of the language definition in an alternative-aware programming language.

Hardware Support

Three important benefits can be realized by using a dedicated external controller instead of mouse and keyboard input for parameter control. First, spatially-multiplexed input enables users to modify multiple parameters simultaneously. Second, with mouse control, tuning is mainly a hand-eye coordination task—with a dedicated control board, it turns into a motor task that leaves the eyes free to focus on the application being tuned. Third, moving the tuning UI to a dedicated controller allows for tuning of interactions that require mouse and keyboard input, *e.g.*, adjusting the rate at which mouse wheel movement magnifies a document.

Our implementation supports a commercially available USB MIDI device [2] with 16 buttons with LED status indicators, 8 rotary encoders (presently not used) and 8 *motorized* faders. The controller transmits input events as MIDI control change messages and receives similar control change messages to actuate sliders and toggle LED feedback. Actuation of the hardware controller is essential for saving and restoring parameter snapshots—without actuation it is impossible to recall saved parameter values and edit them incrementally. To facilitate locating a particular variable's control, the mixer was augmented with a small top-mounted projector which displays parameter names next to the appropriate controls, a technique inspired by Crider et al., [8]. While a projector setup is unwieldy in practice, controllers with embedded text LCDs that can offer the same functionality are commercially available.

DESIGN SPACE & DISCUSSION

The design choices made during the development of Juxtapose represent one particular point in a larger space of tools for explorative programming. In this section, we discuss assumptions made in our current design and highlight limitations of our implementation. Following Fitzmaurice's design space for graspable interfaces [10], we summarize the most salient design decisions in Table 1. This design space is not meant to be exhaustive—it covers the decision points encountered during prototyping and development. Nevertheless, the table suggests additional techniques, such as automatic generation of alternatives, which may be a fruitful area for future work.

Would Designers Really Benefit from Linked Sources?

The efficacy of linked editing in Juxtapose rests on the assumption that interaction designers create multiple alternatives of a common code document, where individual alternatives only differ in parameter settings and small sections of code. Experimenting with code in this manner only covers part of the solution space for a given problem. Different *solution approaches* may be based on distinct implementations. Alternatives as discussed in this paper explore options *within* one particular solution strategy. Are alternative designs related enough in practice to benefit from linked editing and tuning?

Beyond evidence from our formative interviews, the book Flash Math Creativity [27] provides detailed examples of source code experimentation by professionals: 15 Flash designers share how they create computational designs in 56 projects. Each project starts from a single idea, *e.g.*, animating geometric grid structures. The designers then show how they modified the initial source to explore the design space.

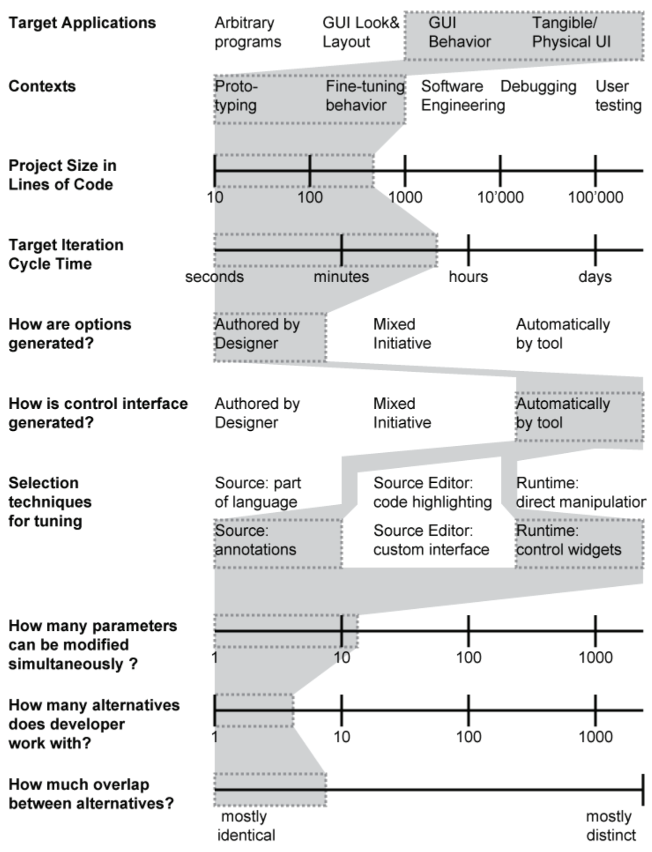


Table 1. Juxtapose design space. Choices implemented by Juxtapose are shown with a shaded background.

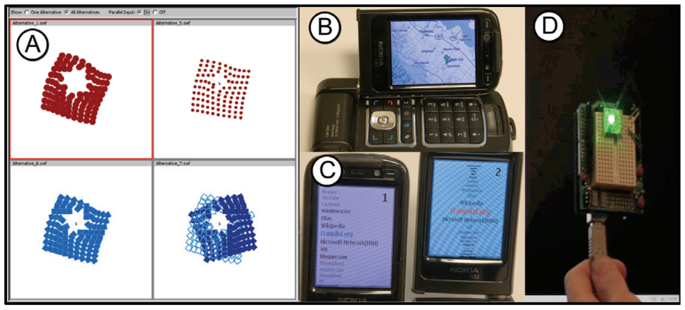


Figure 7. Example prototypes built with Juxtapose. **A**: Reactive grid animation, shown with parallel input. **B**: Map navigation on a phone. **C**: Alternative fisheye menus on two phones in parallel. **D**: Smart RGB LED fading on a microcontroller.

12 of 15 designers showed multiple alternatives for their projects (mean: 10.2 alternatives per project; range: 3 to 23). The difference between these alternatives is usually small: a change to a line of code to load different graphics from a library, alterations to parameter values, or substitutions of function calls.

Is Tuning of Numbers and Booleans Sufficient?

Juxtapose's runtime tuning focuses on direct manipulation of `Boolean` and `Number` types. Would designers benefit from more expressive abstractions and additional functionality in the tuning interface?

An underlying assumption in this work is that developers both produce the application and tune it. If they desire a more complex mapping, *e.g.*, a logarithmic parameter scale, they may express this mapping in the source. Locating additional functionality in the source itself may be more useful since logic expressed in the tuning UI is not available when the application is run outside Juxtapose. This assessment changes if alternatives and tuning options are used by a third party, *e.g.*, during participatory design sessions. In this case it would make sense to imbue the runtime interface with more flexibility to let users express a more complete set of modifications without editing the program source, *e.g.*, by providing rich widgets for commonly used complex data types such as color or coordinates.

Are Code Alternatives Enough?

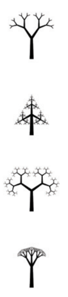
Perhaps the most important limitation is that Juxtapose does not offer support for managing multiple alternatives of graphical assets. Interface design is concerned with both look *and* feel — graphics *and* behavior. We believe Juxtapose is a first step towards an integrated authoring environment that offers management of alternatives across graphics and code. We leave research on such a hybrid environment to future work.

USER EXPERIENCES WITH JUXTAPOSE

To evaluate the authoring approach embodied in Juxtapose, we built example prototypes using the tool (see Figure 7) and conducted a summary usability study of Juxtapose for desktop applications. We recruited 18 participants, twelve male, six female. Participants were recruited from undergraduate and graduate students with HCI experience. Their ages ranged from 20 to 32 years. All but one participant had at least working knowledge of procedural programming and all had at least some expertise in interaction design.

Study Setup

Evaluation sessions lasted approximately 75 minutes. Participants were seated at a workstation with mouse, keyboard and MIDI controller. After a demonstration of Juxtapose, participants were given three tasks. The first task was a warm-up exercise to modify a grid animation reacting to mouse movement, adapted from Flash Math Creativity [27]. Participants were asked to make changes that required both code alternatives and tuning.



The second task was a within-subject comparison that asked participants to adjust four parameters of a recursive tree drawing routine to match four specific tree shapes. The provided code was also adapted from Flash Math. For two trees, this was accomplished using the full Juxtapose interface. For the other two, participants were given the same editor without the possibility of creating alternatives or tuning. Order of assignment between Juxtapose and control conditions was counterbalanced and a random tree order was generated for each participant.

The third task asked participants to work on the mapping scenario introduced earlier. They were provided with a working ActionScript program that loaded a map containing 28 different layers of information (*e.g.*, land areas, parks, local streets, local street names, highways). Participants were given 30 minutes to create two map navigation alternatives. They were then asked to present their maps to a researcher. Documentation contained examples for how to programmatically change visibility of layers, color and brightness, text size and formatting, and mouse interactions. Participants had to modify and add to these examples to either hardcode design decisions or to set up tunable parameters through callback functions in the source code.

Study Results

In all tasks, all participants properly applied linked and unlinked editing and tuning, with no apparent confusion. Participants commented positively on the ease of adjusting numerical parameters through tuning and the reduced iteration time this permitted. One participant commented that the explicit management of alternative documents improved on their existing practice of "half-hearted attempts to name saved [configurations] with memorable names." Today, designers commonly use layer sets as a technique for composing alternatives in graphics. A participant commented that Juxtapose brings this pattern to interaction design.

Tuning Enables More Parameter Experimentation, Faster

In the tree matching task, participants took an average of 258 seconds (σ: 133 s) to complete the matching in the control condition, and an average of 161 seconds (σ: 82 s) to complete the task with Juxtapose (see Figure 8). This difference was significant (one-tailed, paired Student's t-test; $p < 0.01$). When looking at completion times by tree (Figure 8), a large completion time discrepancy for trees three and four becomes apparent. For these trees, participants quickly narrowed in on the approximate shape but frequently had trouble minimizing the remaining visual disparity when they could no longer reason about how to proceed toward the goal. Participants then often broadened their search in parameter space and diverged from the solution while looking for the right parameters to adjust. We believe that Juxtapose outperformed the control condition here because the penalty for an uncertain, diverging move was much smaller — the result could immediately be observed and corrected.

To quantify the cost of making a change, we investigated how many parameter combinations participants explored. In the control condition, on average, participants tested 2.60 parameter combinations per minute to arrive at matches (σ: 0.93; we counted each execution after changing source as 1 combination). In contrast, using Juxtapose, participants executed the Flash file only once, and generated parameter changes through the tuning interface. Here participants explored 64 combinations on average (σ: 80; we counted each variable change sent to Flash as a tuning event). The external MIDI controller generated many input events and one might contend that our definition of parameter change overestimates the number of perceptually different states explored by users. We note that participants adopted a wide range of tuning strategies — some exclusively typing in numbers in the tuning interface, others using multiple sliders simultaneously. This resulted in a wide spread of parameter changes per minute for Juxtapose (see Figure 9), but even participants at lower end of the histogram explored an order of magnitude more states than the control condition.

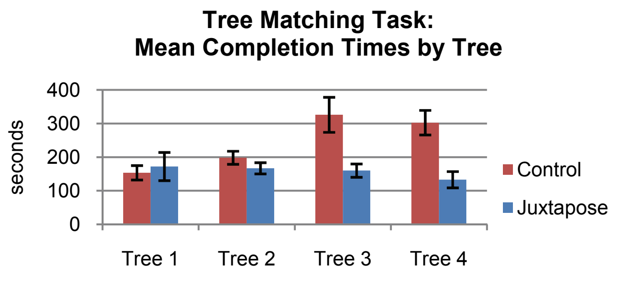


Figure 8. Study participants were faster in completing the tree matching task with Juxtapose than without.

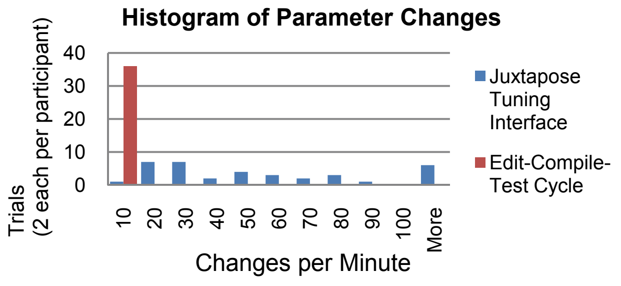


Figure 9. Study participants performed many more design parameter changes per minute with Juxtapose than without.

Alternatives & Tuning Provide Value, At a Price

In our mapping task, many participants began by adding instrumentation code to the provided framework to make map attributes tunable at runtime. While hard-coding design choices into source code would have been easier from a programming perspective, participants spent extra effort to make variables tunable so they could experiment at runtime. Two participants mixed strategies, making some parameters tunable while setting others in code in different alternatives when they were sure about their desired values. For example, one participant hard-coded a higher initial magnification factor in the pedestrian map interface.

Most participants preferred to set the ranges for `Number` variables in source code, not in the runtime interface. Only one participant used the runtime interface for this purpose. A possible explanation is that reasoning about ranges has to do with how a variable is used in the source so participants were more inclined to express ranges there.

Suggestions for Improvement

The map task also uncovered a number of usability shortcomings. In multiple instances, participants closed the runtime window to change a line of code and recompile, discovering that their runtime parameter settings from the last execution were gone. To address this, Juxtapose could automatically save the last parameter values in a snapshot when the runtime window is closed.

Participants also wished for a larger range of variables to access — for the study, only variables declared in the main application's class and variables of the *root* symbol were accessible for tuning. Participants thus had to introduce intermediate variables to influence other graphical objects. It would be preferable to have a "tuning mode" for direct manipulation of all graphical objects, extending ideas introduced in SUIT [26].

Many participants expressed frustration at the lack of search and undo in the source editor. Search could clearly be added. Supporting undo for documents with alternatives is not trivial and an area for future work. Multiple participants also felt that it was overly onerous to properly write the application callbacks that make a design tunable. This can be addressed in two ways. Directly modifying object fields can be handled by making all fields tunable, as suggested earlier. More complex parameter mappings however will still require callbacks: producing these callbacks can be supported through a code generation wizard.

RELATED WORK

Our work is inspired by prior research on design tools for working with alternatives and augmented programming environments for user interfaces.

Tools for Working with Alternatives

The research embodied in Juxtapose was directly motivated by Terry *et al.*'s prior work on tools for creating alternative solutions in image editing. Side Views [31] offer command previews, *e.g.*, for text formatting, inside a tooltip. Parameter Spectrums [31] preview multiple parameter instances to help the user choose values. Similar techniques are now part of Microsoft Office 2007, attesting to the real-world impact of exploration-based tools. Parallel Pies [32] enable users to embed multiple image filters into a single canvas, by subdividing the canvas into regions with different transformations. Since Juxtapose targets the domain of textual programming of interaction designs, its contributions are largely complementary. Unlike creating static visual media, the artifacts designed with Juxtapose are interactive and stateful, which requires integration between source and runtime environments.

Terry also proposed *partials*, an extension to Java syntax that delays assignment of values to variables until runtime [29, Appendix B]. Partial variables list a set of possible values in source code; at runtime, the developer can choose between these values through a generated interface. Juxtapose extends this work by contributing both authoring environment and runtime support for specifying and manipulating alternatives.

Automatic generation of alternatives was proposed in Design Galleries [23], a browsing interface for exploring parameter spaces of 3D rendered images. Given a formal description of a set of input parameters, an output vector of image attributes to assess, and a distance metric, the Design Galleries system computes a design-space-spanning set of variations, along with a UI for structured browsing of these images. Design Galleries require developers to manually specify a set of image features to steer a dispersion algorithm; options are then generated automatically. In Juxtapose, options are created by the designer. Juxtapose makes the assumption that the results of parameter changes can be

	Does evaluation of output require real-time input?	**How are parameter values created?**	**Who creates parameter-to-output mapping?**
Design Galleries	No — output is a static image or a sequence of images.	Generated by dispersion algorithm	Expert specifies for each DG instance
Side Views/ Parallel Pies	No — output is a static image	Mixed initiative: parameter spectrums are auto-generated; designers chooses values	Mixed: image processing library provides primitives; designers compose primitives in Side Views
Juxtapose	Yes, output is a user interface	Designer creates values in code alternatives or tunes at runtime	Developers specify mapping in their source code

Table 2. Differences between the two most related research projects and Juxtapose are based on requirements of real-time input, method of alternative generation, and source of input-output mapping.

viewed instantaneously, while rendering latency motivated Design Galleries. Table 2 shows a comparative overview of Design Galleries, Terry *et al.*'s work, and Juxtapose.

Subjunctive interfaces [21] introduced working with alternatives in information processing tasks. Multiple scenarios co-exist simultaneously and users are able to view and adjust scenarios in parallel. Clip, connect, clone [11] applies these interface principles to accessing web application data, *e.g.,* for travel planning. There are no design tools for creating subjunctive interfaces; only applications that realize these principles in different information domains.

Spreadsheets also inherently support parallel exploration through their tabular layout. Prior research has applied the spreadsheet paradigm to image manipulation [19] and information visualization [6]. Such graphical spreadsheets offer a more complex model of defining and modifying alternatives than Juxtapose's local-or-global editing. Investigating how a spreadsheet approach could extend to interaction design is an interesting avenue for future work.

TEAM STORM [14] addresses management of multiple sketches by a team of designers during collaborative ideation. The system, consisting of individual tablet devices and a shared display wall, allows design teams to manage and discuss multiple visual ideas. Like Terry's work, the system only addresses working with static visual media – interaction can be described in these sketches, but not implemented or tested.

Authoring tools with editable design histories can also support exploration of alternatives since prior decisions can be changed at any point in the process. Design histories preserve flexibility to change the end result, but they still operate on the single document model: only a single artifact is being created. Editable histories have been developed for visual design and information architecture [15, 17].

Development Environments

Juxtapose also relates to tools for developing user interfaces and interactive systems. Amulet [25] is a GUI development environment that features a runtime Inspector [24] to debug interfaces. The Inspector exposes the complete runtime application state and supports debugging of dynamic behavior, such as tracing changes in variable values over time. Juxtapose shares the Inspector's motivation of making internal state visible and modifiable; it also shares the implementation approach of using a library to access runtime features instead of modifying the language compiler or interpreter itself. The Amulet Inspector focuses on *diagnosing* and *correcting problems*; Juxtapose emphasizes *exploring variability* in programs.

Some tools offer "live" coding where all code and data modification happens while the program is executing. JPie [13] is such an environment for Java education; ChucK [35] offers live coding for music synthesis. Juxtapose shares the goal of eliminating edit-compile-test cycles in favor of real-time adjustment. Juxtapose offers less flexibility than live coding languages for editing objects and logic. Conceptually, Juxtapose makes a distinction between a low-level source representation, and a higher-level set of "knobs" used for runtime manipulation. This higher-level abstraction allows for more controlled live improvisation.

The notion of parameter snapshots exists in Isadora [7], a visual dataflow language for multimedia authoring. In Isadora, the parameter sets are predetermined by the library of data processing nodes. In Juxtapose, the programmer can define new variables for tuning in the source. Adobe's Image Foundation Toolkit [4] automatically creates control sliders for scalar parameters in image processing code. In this domain, the entire specified algorithm can be rerun whenever a parameter changes. Juxtapose offers a more general approach that enables developers to control what actions to take when a variable value is changed at runtime.

CONCLUSION AND FUTURE WORK

We have presented a set of techniques to explore and manage alternatives of programmed interaction designs in Juxtapose, a prototyping environment for user interfaces. The insight driving our research is that options have to be managed across *both* source and execution environments. Selective parallel editing linked with an execution environment that is aware of this parallelism enables exploration of code alternatives. Automatic generation of control interfaces and snapshots enable real-time tuning of application parameters. We demonstrated applicability of these techniques for desktop graphical user interfaces, mobile interfaces and physical computing interfaces.

Going forward, it is promising to integrate our research with prior work on graphic alternatives to create an authoring environment for both code and graphics. Another direction worth pursuing is to extend parallel editing and tuning to web applications, which could provide a way to rapidly gather empirical data on user preferences for different alternatives. Large web sites already routinely test alternatives of new features by running controlled bucket experiments: a small percentage of site visitors are exposed to a new proposed feature or layout, and results (time spent on site, purchases made) are compared with the control condition [16]. An interesting an as-of-yet unexplored research question is to what extent such comparative testing with remote users is possible during earlier prototyping stages.

ACKNOWLEDGMENTS

This research was supported through NSF grant IIS-0745320; equipment donations from Intel and Nokia; and a SAP Stanford Graduate Fellowship for Björn Hartmann.

REFERENCES

1 *Arduino Physical Computing Platform*, 2006. http://www.arduino.cc

2 *B-Control Fader BCF2000*, 2007. Behringer. http://www.behringer.com/BCF2000

3 *EMFlash*, 2007. Markelsoft. http://www.markelsoft.com/products/emflash

4 *Image Foundation Toolkit*, 2008. Adobe Systems. http://labs.adobe.com/wiki/index.php/AIF_Toolkit

5 *MTASC: Motion-Twin ActionScript 2 Compiler*, 2005. Motion Twin Technologies. http://www.mtasc.org

6 Chi, E., J. Riedl, P. Barry, and J. Konstan. Principles for Information Visualization Spreadsheets. *IEEE Computer Graphic and Applications.* **18**(4). pp. 30-38, 1998.

7 Coniglio, M., *Isadora*, 2008. Troikatronix. http://www.troikatronix.com/isadora.html

8 Crider, M., S. Bergner, T. N. Smyth, T. Möller, M. K. Tory, A. E. Kirkpatrick, and D. Weiskopf. A mixing board interface for graphics and visualization applications. In *Proceedings of Graphics Interface 2007*. ACM. pp. 87-94, 2007.

9 Drexler, K. E., *Engines of Creation: The Coming Era of Nanotechnology*: Anchor Books. 320 pp. 1986.

10 Fitzmaurice, G. W., H. Ishii, and W. A. S. Buxton. Bricks: laying the foundations for graspable user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. pp. 442-49, 1995.

11 Fujima, J., A. Lunzer, K. Hornbæk, and Y. Tanaka. Clip, connect, clone: combining application elements to build custom interfaces for information access. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*. pp. 175-84, 2004.

12 Gaver, B. and H. Martin. Alternatives: exploring information appliances through conceptual design proposals. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. pp. 209-216, 2000.

13 Goldman, K. J. An Interactive Environment for Beginning Java Programmers. *Science of Computer Programming* **53**(1). pp. 3-24, 2004.

14 Hailpern, J., E. Hinterbichler, C. Leppert, D. Cook, and B. P. Bailey. TEAM STORM: demonstrating an interaction model for working with multiple ideas during creative group work. In *Proceedings of the 6th ACM SIGCHI conference on Creativity & Cognition*. pp. 192-202, 2007.

15 Klemmer, S. R., M. Thomsen, E. Phelps-Goodman, R. Lee, and J. A. Landay. Where Do Web Sites Come From? Capturing and Interacting with Design History. CHI: ACM Conference on Human Factors in Computing Systems, *CHI Letters* **4**(1). pp. 1–8, 2002.

16 Kohavi, R., R. M. Henne, and D. Sommerfield. Practical guide to controlled experiments on the web: listen to your customers not to the hippo. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. pp. 959-67, 2007.

17 Kurlander, D. and S. Feiner, A Visual Language for Browsing, Undoing, and Redoing Graphical Interface Commands, in *Visual Languages and Visual Programming*, S.K. Chang, Editor. Plenum Press: New York, NY. pp. 257-75, 1990.

18 Lawson, B., *How Designers Think: The Design Process Demystified*. 3rd ed: Architectural Press. 352 pp. 1997.

19 Levoy, M. Spreadsheets for images. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. pp. 139-46, 1994.

20 Lunzer, A. Choice and Comparison Where the User Wants Them: Subjunctive Interfaces for Computer-Supported Exploration. In Proceedings of *INTERACT '99: IFIP Conference on Human-Computer Interaction*. pp. 474-82, 1999.

21 Lunzer, A. and K. Hornbaek. Subjunctive interfaces: Extending applications to support parallel setup, viewing and control of alternative scenarios. *ACM Transactions on Computer-Human Interaction.* **14**(4). pp. 1-44, 2008.

22 Mackay, W. E., C. Appert, M. Beaudouin-Lafon, O. Chapuis, Y. Du, J.-D. Fekete, and Y. Guiard. Touchstone: exploratory design of experiments. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. pp. 1425-34, 2007.

23 Marks, J., B. Andalman*, et al.* Design galleries: a general approach to setting parameters for computer graphics and animation. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. pp. 389-400, 1997.

24 Myers, B. A., A. Ferrency, R. McDaniel, and R. Dannenberg, *Debugging Interactive Applications*, Carnegie Mellon University 1996. http://www.cs.cmu.edu/~amulet/papers/debugpaper.pdf

25 Myers, B. A., R. G. McDaniel, R. C. Miller, A. S. Ferrency, A. Faulring, B. D. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Transactions on Software Engineering* **23**(6). pp. 347-65, 1997.

26 Pausch, R., M. Conway, and R. Deline. Lessons learned from SUIT, the simple user interface toolkit. *ACM Transactions on Information Systems.* **10**(4). pp. 320-44, 1992.

27 Rycroft, S., ed. *Flash Math Creativity*. 2nd ed. Friends of Ed. 264 pp., 2004.

28 Schrage, M., *Serious Play: How the World's Best Companies Simulate to Innovate*: Harvard Business School Press. 278 pp. 1999.

29 Terry, M., *Set-Based User Interaction*, Unpublished Ph.D. Thesis, Georgia Institute of Technology, Computer Science Department, 2005.

30 Terry, M. and E. D. Mynatt. Recognizing creative needs in user interface design. In *Proceedings of the 4th conference on Creativity & Cognition*. pp. 38-44, 2002.

31 Terry, M. and E. D. Mynatt. Side views: persistent, on-demand previews for open-ended tasks. In *Proceedings of the 15th annual ACM symposium on User interface software and technology*. pp. 71-80, 2002.

32 Terry, M., E. D. Mynatt, K. Nakakoji, and Y. Yamamoto. Variation in element and action: supporting simultaneous development of alternative solutions. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. pp. 711-18, 2004.

33 Tohidi, M., W. Buxton, R. Baecker, and A. Sellen. Getting the right design and the design right. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*. pp. 1243-52, 2006.

34 Toomim, M., A. Begel, and S. L. Graham. Managing Duplicated Code with Linked Editing. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*. pp. 173-80, 2004.

35 Wang, G. and P. R. Cook. On-the-fly programming: using code as an expressive musical instrument. In *Proceedings of the 2004 conference on New interfaces for musical expression*. pp. 138-43, 2004.