Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors

Amy J. Ko and Brad A. Myers
Human-Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213 USA
{ajko, bam}@cs.cmu.edu

ABSTRACT

Recent advances in programming environments have focused on improving programmer productivity by utilizing the inherent structure in computer programs. However, because these environments represent code as plain text, it is difficult and sometimes impossible to embed interactive tools, annotations, and alternative views in the code itself. Barista is an implementation framework that enables the creation of such user interfaces by simplifying the implementation of editors that represent code internally as an abstract syntax tree and maintain a corresponding, fully structured visual representation on-screen. Barista also provides designers of editors with a standard text-editing interaction technique that closely mimics that of conventional text editors, overcoming a central usability issue of previous structured code editors.

Author Keywords

Structured editors, programming environments. end-user software engineering.

ACM Classification Keywords

D2.6 Programming environments: *Interactive environments*; H5.2 User interfaces: *Interaction styles*.

INTRODUCTION

Programming environments are the primary user interfaces for millions of professional and end-user programmers' work. In recent years, this observation has led to many efforts to improve programmer productivity with new tools. One tool that exemplifies this effort is the Eclipse environment: by incrementally compiling source files as the programmer edits them, it can offer semi-immediate feedback about errors, quick fixes for common problems, code refactoring tools, and improved searching and navigation support.
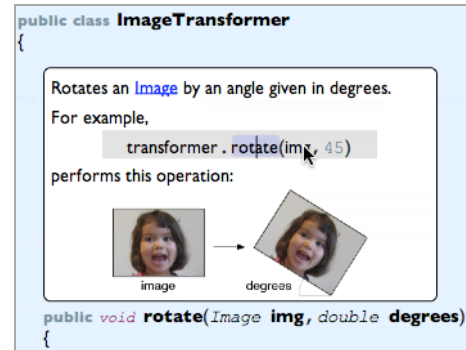
Figure 1. A media-rich annotation of a Java method.

However, many other potentially useful tools are difficult or impossible to implement in environments like Eclipse that visually represent code as rows of plain text. For example, consider the media-rich annotation portrayed in Figure 1; plain text editors would be forced to show such annotations as raw HTML source, hiding contextually relevant information. Other examples include alternative views of code: rather than just text, expressions could be pretty-printed to improve readability [1] and complex operations on data structures could be animated to improve understanding [7].

Although it is more feasible to implement these ideas in structured code editors, such as those developed in the 1980's [13, 16, 18] and more recently with powerful frameworks such as Harmonia [3] and Proxima [17], none of these have support for embedding such views in code, forcing any contextually relevant information about code to be placed out of context, in separate windows and dialogs.

In this paper we present **Barista**[1], an implementation framework that overcomes these limitations by offering data structures, algorithms, and interaction techniques for creating sophisticated code editors like the one in Figure 1. The goal of Barista is to make it possible to implement novel tools and interactions that improve the usability, user experience, and utility of code editors. In fact, the annotation support shown in Figure 1 was implemented in just a few lines of code using the Barista framework. This

---

[1]   **B**asic **A**bstractions for **R**apidly **I**mplementing **S**tructured **T**ext-editing **A**pplications. A *barista* is makes coffee; our example editor helps make Java code.

Most up-to-date version: 06/22/2021

support was built on top of a basic Barista Java editor, which is written in Citrus [11]. Furthermore, unlike previous structured code editors, all Barista editors come with a standard text-editing interaction technique that is quite similar to the one used in conventional text editors. This is achieved by fluidly changing the code between structured and unstructured text, while presenting a consistent visual representation of code on-screen. Combined, these features result in an implementation framework that is simple and powerful for editor designers and may enable the creation of more usable and useful code editors for users.

In the following sections, we provide an overview of related work on code editors that utilize structure. We then describe the Barista framework in detail and provide numerous examples of the types of tools that the framework enables. We conclude with a discussion of Barista's design tradeoffs and limitations relative to other frameworks.

RELATED WORK

Since the Cornell Program Synthesizer [18] in the late 1970's, there have been many efforts to develop usable code editors that utilize the inherent structure in computer programs. While there are several instances of structured editors from the past 30 years [8, 13, 16], we will focus our discussion on implementation frameworks that have been designed for *creating* these editors, highlighting tradeoffs which have made it difficult to implement embedded tools and visualizations like the one portrayed in Figure 1.

One of the first frameworks for developing structured code editors was the GNOME project [4]. Given a language grammar, most of a GNOME editor could be generated automatically. The editors were largely based on a model-view-controller architecture, where the models were the abstract syntax trees representing the program and the views were textual. However, the editors provided no support for editing a textual representation of the code; instead, they used a generic menu-based interaction technique for structurally modifying trees. The MacGnome project relaxed this by allowing small sections of code to be temporarily converted into plain text in a separate editing mode [11]. Although GNOME editors did support multiple views and languages, their visual representation of code was limited to text with little support for customization.

Proxima is a more recent implementation framework for developing structured editors consisting of five "layers": document, extended document, layout, arrangement, and rendering. The editor designer's job is to define the document's representation for each layer, and supply mappings between all layers that will be edited by the end user. For example, end users may edit the document at the "layout" level (which is equivalent to editing the text of a program), or they may edit at the document level (such as inserting a declaration into a list). While the framework provides a great deal of flexibility in the design of an editor

and its interaction techniques, it does so by placing a great burden on the editor designer to implement all of these layers, as well as to carefully design custom interaction techniques for each editable layer. To complicate matters, implementing the mappings between layers requires the editor designer to learn multiple languages.

Another example is Eclipse (www.eclipse.org), which has one goal of providing abstractions for implementing sophisticated code editors. Unlike GNOME and Proxima, Eclipse's framework is largely text-based. Consequently, Eclipse editors provide little flexibility in defining novel views of code (which is one reason why few Eclipse plug-ins actually do). Furthermore, while Eclipse provides several APIs for implementing the abstract syntax trees of the target language, the connections between these trees, the parser, and the textual representation of the trees is very weak. As a result, it requires considerable effort to write code to perform these mappings when implementing tools.

The Harmonia [3] project, another recent effort to develop a framework for developing sophisticated programming environments, has its strengths in providing a general framework for incrementally lexing and parsing text using a language grammar. Editor designers can easily create text editors that utilize these services, but there is little support for defining custom views of code other than variations in font size and color. Harmonia's predecessor, Ensemble, used Proteus [5], a flexible constraint-based layout presentation framework. However, it required the design of custom interaction techniques for each new view. Also, the language used to define views of code was different than the language used to implement the rest of an editor.

Although all of the frameworks discussed in this section are limited in their ability to support the type of interactive tools and visualizations we have proposed, they do have several benefits that Barista does not emphasize. We will further discuss these tradeoffs in our discussion.

THE BARISTA IMPLEMENTATION FRAMEWORK

To simplify the implementation of code editors that enable novel visualizations and tools, Barista combines ideas from other editors and implementation frameworks, as well as many new ideas, and distills them into a small number of simple abstractions and implementation mechanisms.

Throughout this paper, we will describe the framework in the context of the prototype Java editor seen in Figure 1 and Figure 2. This editor allows users to create code by either typing it with the keyboard, through drag and drop of code templates, or using auto-complete menus that contain the structures and as well as variable and method names that may be legally inserted at the caret. The editor provides immediate feedback about errors as well as several other features that we will describe below. All of these features were implemented in about 2000 lines of Citrus code, using the Barista framework and Citrus UI toolkit [11]. The framework itself is also about 3000 lines of Citrus code.
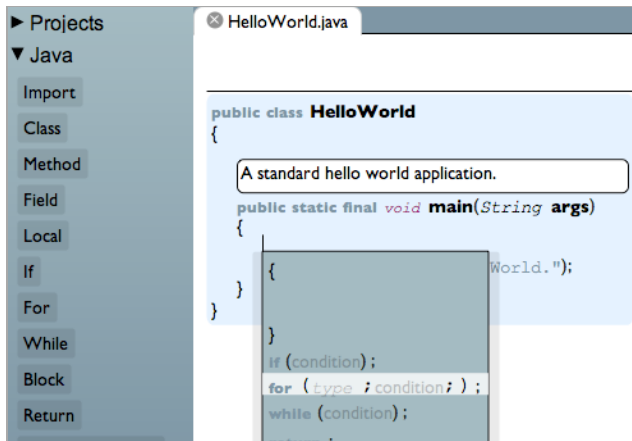
Figure 2. A Java editor, created with Barista, supporting textual editing, auto-completion and drag and drop.

An Overview

Although more powerful architectures have been proposed, such as Proxima's layered architecture [17], for simplicity's sake, Barista is based on the model-view-controller architecture with which many programmers are already familiar. The *model* in a Barista editor consists of the abstract syntax tree that is modified during an editing session, which consists of *structures* and *tokens*. For example, Java's structures include a variety of declarations, statements and expressions, and several kinds of tokens, including identifiers, separators, literals, and keywords. The *view* in a Barista editor is a tree of interactive views (just like the trees found in conventional user interfaces). This view tree mimics the structure of the model. *Controllers* are implemented using the event-handling constructs offered by the Citrus programming language [10].

To illustrate, the models and views used to represent a Java `IfStatement` are shown in Figure 3. Each of the structures in the model is represented by a container view, which is responsible for arranging and displaying the views of the structures contained in its model. For example, the `IfStatementView` contains a `LessThanView` and a `ReturnStatementView`, and views of the if-keyword and parentheses tokens. Each of these tokens consists of the token text and its trailing white space. These are visually represented with two text fields, which act as the interaction points for all textual edits. Because Barista uses a conventional view tree, designers have the same control over the layout, appearance and interaction techniques of views, as they do with any conventional user interface.

```
an IfStatement is a Structure that
   has IfKeyword     if
   has LeftParen     left
   has Expression    condition
   has RightParen    right
   has Statement     thenStatement
   has ElseKeyword   else
   has Statement     elseStatement
 .
```

Figure 4. The Citrus class for a Java if-statement, representing `if (Expression) Statement [else Statement]`.
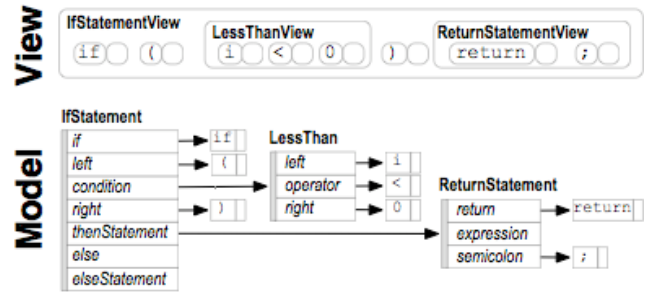

Figure 3. The models and views represented in memory for the structures and tokens in a Java if-statement. Each structure has a set of properties that point to structures and tokens. Each token has text and white space.

Structures and Structure Views

To create a Barista editor, the designer defines each of the structures in the target language as a subclass of `Structure`. This roughly corresponds to the non-terminals in the language's grammar. For example, Figure 4 shows the Citrus class that we defined to represent Java's grammar for an if-statement. The class declares that an `IfStatement` consists of seven properties; some are structures, like the `Expression` named `condition`, and others are tokens, like the `LeftParen` named `left`.

To be able to parse each kind of structure in a language, each class must have a grammar. By using reflection to inspect the properties type, initial values, and order, Barista is often able to infer a suitable grammar for a Citrus class. However, because grammars must often be modified to be parsable, it is sometimes necessary for designers to explicitly provide one.

Once a `Structure` is defined, the designer subclasses `StructureView` to represent it visually. To illustrate, consider the `IfStatementView` class, shown in Figure 5.

```
an IfStatementView is a StructureView that

 refs IfStatement model = ?
 has Real width <- (this rightmostChildsRight)
 has Real height <- (this tallestChildsHeight)
 has List<View> children = [
   (model.@if toView)
   (model.@left toView)
   (model.@condition toView)
   (model.@right toView)
   (model.@thenStatement)
   (an ElseView)
 ]

an ElseView is a View that

 has Bool hidden <-
   (model.elseStatement is nothing)
 has Bool scale <- (if hidden 0.0 1.0)
 has List<View> children = [
   (model.@else toView)
   (model.@elseStatement toView)]
 .
```

Figure 5. The Citrus class for a view of a Java if-statement.

The view declares a property named `model`, which is a reference to the `IfStatement` that the view represents (the `?` declares that the model is required at the time of instantiation, whereas the other properties have default values). The next two lines use the `<-` operator to *constrain* the view's width and height to the expressions shown. The next property, named `children`, is a list of all of the views contained in the `IfStatementView`. For example, the 3rd child is a view of the `IfStatement`'s `condition`. The expression (`model.@condition toView`) results in a view of the `condition` (for example, the `LessThanView` shown in Figure 3). The `@` in the `toView` expression gets the *property* that points to the condition. This way, the view can listen for changes to this property's value, and upon a change, automatically generate a view of the property's new value, freeing the designer from having to write code to keep views up to date. Because the else statement is optional, we define an inner class named `ElseView` that contains a view of the `else` keyword and else statement. A constraint on `hidden` hides the view when the `IfStatement`'s `else` property statement is empty.

The default layout of the children in a `StructureView` is the same flow layout used by conventional text editors. This is implemented by constraining the position of a structure or token view to the position after the last character in the view of the previous token in the abstract syntax tree. This can easily be overridden by using any of the other standard layouts provided by the Citrus UI toolkit, or by defining new layouts or constraints.

The view defined in Figure 5 is fairly basic, but several extensions can be added in just a few lines of code. For example, a "collapsed" flag could be added, and a property change listener could hide the body of the if-statement when true. To design an alternative view, such as a table of condition-action pairs, would take about the same number of lines of code as in Figure 5, but would use a different layout and would omit the `if` and `else` keywords. The designer could then provide a way to toggle between the two. We give an example of this later.

Tokens and Token Views

In addition to defining structures and corresponding structure views, editor designers must also specify each type of token in the target language as a subclass of `Token` and specify regular expressions that define its legal strings and white space. For example, instances of the `IfKeyword` class, used in Figure 4, must always match the string "if" and may have arbitrary whitespace.

Barista provides the `TokenView` class, which represents a token as two text fields that contain the token's text and whitespace. This class implements a conventional text editing interaction technique and provides auto-complete menus that are generated automatically using the Barista parser and designer-supplied grammars. Editor designers can subclass this view to customize its size, font, color, layout and behavior. For example, in our Java editor, each

token of a particular type has a different font and color. Constraints are used to change the font depending on the structure that contains the identifier; for example, as seen in Figure 2, an identifier that appears as a name of a class has a larger font size than an identifier in an expression.

The `TokenView` class' text editing interaction uses a variant of incremental parsing algorithms [19]. When the user types a character into a token's text or white space:

1. If the modification to the token results in a string that complies with the token's regular expression, the modification is allowed and we are done.

2. Otherwise, the token and its immediately adjacent tokens need to be re-tokenized. This results in a sequence of one or more new tokens. If tokenization fails, the characters are placed in a special invalid token, which will be re-tokenized when it is modified.

3. If the tokenization succeeds, the part of the abstract syntax tree affected by this token modification is converted into a list of tokens. Structures that were not affected, but were part of an affected tree, remain structured, and are inserted into the list of tokens as "structural" tokens, and handled specially by the parser. For example, if a Java block needed to be parsed, none of the unaffected statements inside the block would be reparsed.

4. The list of tokens is parsed, reusing any structural tokens found in the process. The resulting structure replaces the old structure in the tree, which automatically updates the view in the editor.

To illustrate this process, consider the change illustrated in Figure 6. When the user types the + into the text field representing `a123`, the parser is invoked by the token view. Because `a+123` is an invalid Java identifier, it is retokenized into the tokens [`a`, `+`, `123`]. If the user had instead changed `a123` to `ab123`, it would still have been a
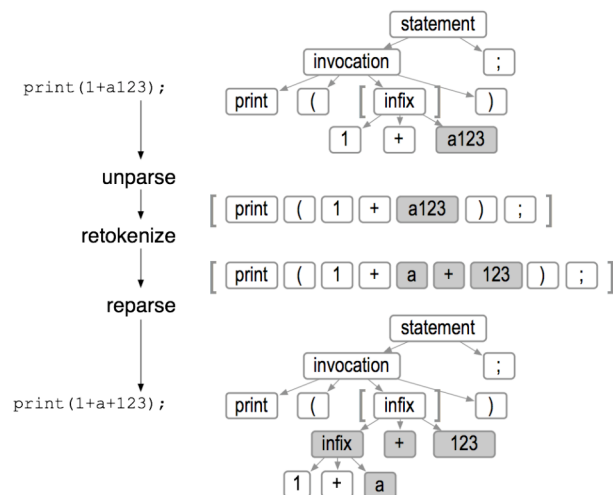

Figure 6. An example of Barista's incremental parsing. The grey boxes indicate data that was modified or created.

valid identifier and the token view would have allowed the change. The affected structure—the common ancestor of the structures to the left and right of the affected tokens in the tree—is "unparsed" into a sequence of tokens. In this case, the structures left and right are the parentheses, whose common ancestor is the method invocation. The modified token is replaced by the new tokens, and the list is then parsed. While parsing, the parser determines that it can reuse the existing invocation structure, and just replaces the first expression in the invocation's argument list with the newly created infix expression. The rest of the program is unaffected. The token view then replaces old invocation with the new, automatically updating the view. This process is transparent to both the user *and* the editor designer.

We can avoid incremental parsing in some cases by using information encoded in the abstract syntax tree. For example, in our Java editor, changing = to += only requires a single token to be updated, because both tokens are instances of the `AssignmentOperator` class. If the only change to a structure is the insertion of a new structure into a list, for example, when a statement is inserted into a block, we avoid reparsing the list and instead just structurally insert the new structure.

Barista provides several other mechanisms by which incremental parsing may be invoked. By default it is invoked after every character typed. Another approach is to reject any modification to a token that fails to parse. This is equivalent to what traditional structured editors do, which only allow valid transformations to the tree. The implication for the end user however is that many desired edits would only be possible through a series of structural edits, which typically require far more operations than would be needed in textual editors. Another choice is to convert the modified tokens to a single invalid token (as mentioned in step 3 above). The token view can then later tokenize and incrementally parse upon one of several events: for example, after the programmer has been idle for a certain amount of time or after the user leaves the focus of the token view's text field. Yet another approach is to add error productions to the language grammar (a standard strategy in implementing parsers), which places the unparsable tokens into a special structure. An example of this in our Java editor is shown in Figure 7. The advantage of this is that this interactive flexibility may be added to structures that are modified frequently (such as expressions and statements), and not added for high-level structures such as class declarations. We offer all of these options, not only for performance reasons, but because the appropriate immediacy of feedback differs between languages, language constructs, user populations, and individuals. Our
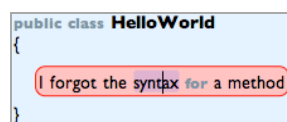
Java editor uses different strategies for different structures. For example, error productions were added for expressions and statements, whereas syntax errors are disallowed on class declarations, which are usually created via drag and drop or menus.

Barista's parser is language independent and is executed by interpreting the grammar supplied by the editor designer or automatically generated from the Citrus classes. The parser can determine what tokens and structures may follow a token, which is used to implement the auto-completion shown in Figure 2. For example, if the parser encounters the tokens [`if`, `(`], it knows it can safely auto-complete the list to [`if`, `(`, `expression`, `)`, `statement`]. These options are exposed to the designer as simple flags.

Text Caret Navigation
Evidence from a study of programmers' low-level text editing strategies [9] suggests that programmers rely heavily on the keyboard for code navigation. Therefore, Barista's `TokenView` class provides built-in navigations that directly mimic those in text-editors. Pressing left or right at the edge of a token navigates the text caret to the previous or next token's white space, and navigating at the edge of white space similarly navigates to the previous or next token in the model. These are implemented in two lines of code using `Token`'s support for determining adjacent tokens in the tree. Pressing up or down finds the horizontally nearest token view in the row above or below the token view that currently has focus, where a row is defined as the sequence of tokens delimited by line breaks.

Copy and Paste
In addition to allowing for structural selection (for example, choosing a whole block, method, or statement), Barista also provides built-in support for selecting an arbitrary sequence of tokens, just like conventional text editors. When a selection is copied and pasted into a token view, Barista attempts to copy as much structure as possible, and then reparses as necessary. Barista also provides interaction techniques for making structural selections with the keyboard, by using shift with the cursor keys. For example, when the caret is in the view of `var` in the expression `print("" + var)`, the user can type shift-up-up to select the whole expression, and then shift-down-down to change the selection back to `var`.

Serialization
By default, Barista saves code and its associated metadata (such as the method header in Figure 1) as XML. Barista can also use the target language's grammar to generate syntactically formatted plain text. In this case, any information associated with the code and any unparsable code, such as that in Figure 7, is stored as XML inside the target language's comment syntax. This allows Barista to recover the information later, while providing backwards-compatibility with widely used text processing tools.

Figure 7. A list of unparsable tokens in a Java class.

EXAMPLES

In this section we describe the implementation of several tools and interaction techniques, in order to demonstrate Barista's simplicity and expressiveness.

Live, Embedded Meta-Data

There are a myriad of ways in which programmers have used textual comments in programs to store information about code. For example, JavaDocs encode specifications, "TODO" tags encode reminders, and programmers even draw diagrams in comments using "ASCII art", where lines, shapes and arrows are approximated with ASCII characters. Also, Latoza et al. have found that developers often take pictures of whiteboard diagrams that explain the rationale behind code, but they are unable to put them in places where others can find them [12]. Unfortunately, text is a very restricted medium in which to author and represent such information.

In Figure 1 we portray a more structured, visual alternative to displaying such metadata, which allows for many kinds of rich annotation. This includes a prose description of the method that includes a hyperlink to the declaration of the `Image` class, as well as a diagram portraying an example of the operation. The example code in the annotation is editable and included in searches and refactoring operations, keeping the example code up to date. Implementing this view required adding a property to our `MethodDeclaration` class to point to a block of HTML (supported by the Citrus UI toolkit) and modifying our `MethodDeclarationView` to show the HTML above its header. We are not aware of any other code editors that provide such a capability.

Highlighting Name Resolution Errors

One obvious requirement for a code editor is to give the user immediate feedback about errors in the code. For example, each identifier token in a Java program (other than those in a declaration), must resolve to a declaration in the token's scope; if it does not, the user should be notified so that they may correct the problem. The feedback in our editor, seen in Figure 8, places red underlines beneath identifiers that cannot be resolved. To provide this feedback required just a few lines of code. We added a `binding` property to each identifier token, which is constrained to the result of a function that resolves a name based on Java's name resolution rules. Then we added a `LinePaint` to the foreground of the `IdentifierView`:

```
has List<Paint> foreground = [(a LinePaint
  y1=1.0 y2=1.0 alpha<-
    (if (model.binding is nothing) 1.0 0.0))]
```

The `y1` and `y2` parameters specify that the line appears at the bottom of the identifier (the x-positions are at the left and right of the view by default). When changes happen in the properties on which the binding property depends, the binding is updated, automatically updating the alpha transparency of identifier view's line if on-screen.

Balancing Delimiters

Balancing parentheses, braces, and brackets is a standard feature of modern code editors, and it was very easy to implement in our Java editor. Our implementation, portrayed in Figure 8, involved adding the following code to our `PairedDelimiterView` class, which is the base class of the views of Java parentheses, braces, and brackets:

```
has List<Behavior> behaviors = [
  (a Behavior event=(a View.ReceivesFocus)
    action='(do
      (if ((model. pair is nothing) not)
        ((model.pair getView).foreground append
        JavaStyle.tokenFocusPaint))
  )]
```

This code states that when this delimiter receives focus, the focus paint, `JavaStyle.tokenFocusPaint`, should be added to the view of the other delimiter in the pair. Each paired token keeps a pointer to its matching token in the property named `pair`. If `pair` points to something, its view is retrieved, and the focus paint is appended to its foreground paint list. A similar event handler is added to remove the focus paint when the focus is lost.

Dragging Code with the Keyboard

The results of an in-depth study of programmers' code editing strategies led to many design ideas for novel "micro-level" refactorings of code [9], which may help programmers make low-level modifications to code more efficiently. For example, programmers frequently moved a statement out of a block into the containing block by moving the statement or the delimiter, sometimes using Eclipse's support for swapping lines of text. Unfortunately, when performed textually, these modifications involved a cumbersome number of selections, cuts, pastes, and caret movements, and when statements or expressions broke across lines, were often performed incorrectly.

As an alternative, we implemented interaction techniques for "dragging" statements and delimiters with the keyboard by holding down a modifier key and using the arrow keys. To implement these, we added keyboard event handlers to the left and right braces and to the base statement view class to respond to control-up and control-down. Because all of the statement views in our editor inherit their event handlers from our `StatementView` class, the user can invoke this command from any token in a statement and the event will passed up the view hierarchy until it is handled by the statement view. To handle the down event, it evaluates `(model getNext Statement)`, which traverses the abstract syntax tree in search for the next structure of type
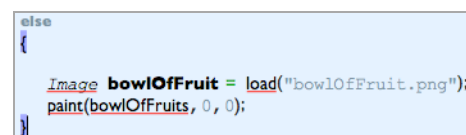

Figure 8. Visual feedback about name resolution errors and delimiter pairs, both implemented in just a few lines of code.

Figure 9. On top, a more readable pretty-printed view of the distance formula. On bottom is the more editable view that is shown when the caret is in the expression.

`Statement`. If it finds one, the event handler removes its model from the block that contains it with the method call `(model.owner.statements remove model)` and then inserts its model into the block that contains the subsequent statement using the expression `(next.owner.statements insertAfter next model)`. The Citrus UI toolkit then automatically handles the necessary updates to the views.

Situated, Task-Appropriate Views

It has long been known that "pretty-printing" computer programs can improve the readability and comprehension of code [1, 14], but traditionally such rendering has been done separate from a programmer's code editor. Even in editors that are capable of rendering such views embedded in code, the users must learn entirely new interaction techniques for editing text that is laid out in unique ways.

Barista enables editors to situate more readable views in the code and switch to more editable versions when users need to edit them. For example, consider the pretty-printed expression shown in Figure 9 and the "match forms" design shown in Figure 10 (a less error-prone alternative to textual representations of logical expressions [15]). When viewed, both of these types of expressions appear pretty-printed, but when the user moves the text caret into any of the tokens of the expression, it changes to a more editable, textual representation. The user can also optionally toggle between views using a keyboard shortcut. Both of these alternative views were implemented just like any other view, but with custom layouts and appearance. To implement these interactions, a `readable` flag was added to all expression views, and each expression view has two children: its readable view and its editable view. The `hidden` flag of each is constrained such that it is shown when `readable` has the appropriate value, and hidden otherwise. The keyboard shortcut simply toggles `readable`.

A Focus + Context View of Statement Blocks

Most modern code editors allow programmers to "fold" or collapse blocks of code to ease navigation and improve programmer's awareness of the surrounding context in a file. One problem with this interaction is that when a programmer wants to view part of the surrounding context, they either have to hover over the collapsed code to show a tool tip, possible occluding relevant code, or expand the collapsed code, possibly moving relevant code off-screen.

Figure 10. On top, a "match form" representation of a Boolean expression [15]; on bottom is the more editable view that is shown when the caret is in the expression.

An alternative, portrayed in Figure 11, allows the user to double-click in the white space of any block in the code to scale the contents of the block to half their normal size. This way, the surrounding context is still legible, but uses less space. To implement this, each block simply responds to a double-click by toggling a `collapsed` flag. Each block's scale is then constrained to the expression (`if collapsed 0.5 1.0`). A simple extension could allow users to control-drag in the white space to scale the code to *any* size. This would be implemented by responding to the drag event, and instead of constraining the scale of the block, just modifying it based on the user's initial drag position.

Another interaction could elide code by replacing it with "...", revealing the surrounding context. This would be implemented in the same way as the views in Figures Figure 9 and Figure 10. Another alternative view of code could summarize a user's location in a file by showing the structure surrounding the text caret's location. For example, if the caret was in an if-statement's block, the view might show the file, method, and if-statement condition that the caret is in, hiding the rest of the surrounding code.

Figure 11. A focus + context interaction that allows users to partially collapse blocks of Java code.
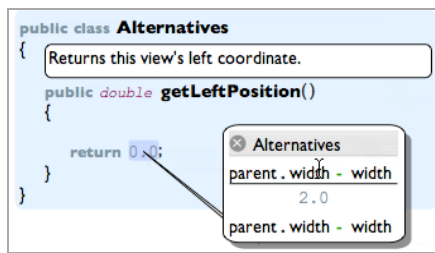
Figure 12. A tool for managing alternative expressions, helping users track and explore different solutions.

Tracking Alternative Expressions

Languages such as Lisp and Python are frequently noted for their support for exploratory programming, the notion that programmers should be able to rapidly test alternatives. While many languages today would claim to support exploratory programming, no language or environment has made it easy to keep track of alternatives that have been explored in the code. Instead, programmers must comment out code they want to temporarily remove, potentially losing any useful information stored in the structure, cluttering the text, and often causing errors [10].

The novel tool in Figure 12 offers a place to keep track of alternative expressions. Unlike code that is excluded with comments, the code in these structures can still be edited, still has support for auto-completion, can still be searched for, and can still be changed by refactoring tools. To implement this, we simply added an optional list of alternative expressions to each Expression structure, and defined a draggable, closeable view for this list, seen in Figure 12, to display these alternatives when the expression receives focus. The user can create an alternative expression by typing a keyboard shortcut in an expression, which replaces the expression with an empty expression, and places the old expression in the alternatives list. The user can swap in alternatives in this list by selecting them from the list, and can remove them by backspacing.

DISCUSSION

All code editors must aid users in manipulating the abstract syntax tree that represents a program written in a particular language. Beyond this basic requirement, there are a number of dimensions in which we can compare code editor implementation frameworks and the editors that they help to implement.

Implementation Effort

A central contribution of Barista is the simplicity of implementing editors and the novel tools and interaction techniques that we have described. Obviously, the small examples we presented in the previous section cannot demonstrate the complexity of implementing more refined and robust tools like those found in commercial systems. Such comparisons are warranted, but difficult to perform without extensive use of both implementation frameworks.

Nevertheless, there are several inherent differences between the Barista and other frameworks that suggest some advantages. For example, because Barista's implementation language is Citrus, a dynamic language with language-level support for constraints, reflection, and event handling, many common behaviors such as updating views and manipulating abstract syntax trees are more easily expressed and require less code. Furthermore, many systems, such as Proxima [17], require designers to learn different languages for defining models and views of code in different layers, whereas Barista only requires one. While multiple languages may have the advantage of specific notations and syntax for specific tasks, this additional requirement raises issues about how the different languages may interact to affect the document. As seen in many of the examples in this paper, in Barista editors, it is straightforward to manipulate the model from the view.

Language Independence

There are generally two ways that code editor implementation frameworks can be language independent. One is by providing universal interaction techniques, which users can learn once and use for in any editor, for any language. Plain text code editors do exactly this, and structured code editors should do the same. Barista succeeds in this regard by offering a standard text-editing interaction technique and even standard menu-based and drag-and-drop interactions. Other structured code editors have either provided universal, but inflexible interactions (such as menus and drag-and-drop interactions exclusively), or no standard interaction techniques at all.

The second way in which code editor implementation frameworks can be language independent is by supporting the creation of editors for *any* type of language. For example, Harmonia [3] provides powerful support for generating efficient and flexible parsers and lexers for any *textual* language, but it does not support *visual* languages such as LabView [6], which are also widely used. Barista can support both because of its flexible presentation model. We have begun to implement editors for a variety of other languages including Lisp, Citrus, and Python.

Flexibility of Interaction Techniques

Another important dimension of editor frameworks is the flexibility of the interaction techniques that they support. This can be characterized in terms of two extremes. The first extreme is the approach of purely parsing editors, which *derive structure from presentation* (the presentation in this case being the text). All textual code editors use this approach. One benefit of this approach is that the interaction techniques for editing sequences of characters (text carets, selections, copy and paste, etc.) is a prerequisite for basic computer use, so every computer user has mastered them. In purely textual editors, however, users must learn a considerable number of syntactic rules to type correct code so that the compiler will be able to parse it (a long-standing and well-understood problem).

The opposite extreme is that of a pure structured editor, which involves *deriving presentation from structure* (generating text from structure). Although pure structured editors can prevent syntax errors entirely, they require users to learn far more operations than those in a text editor. Furthermore, structured editing interaction techniques have traditionally been less efficient and more inflexible than keyboard-based interaction techniques, requiring a mouse to navigate through lengthy menus of valid structures and to drag and drop structures from toolbars. Although Barista does derive its presentation from structure, it overcomes these limitations by including parsing techniques that treat the structure as if it were textual, while at the same time also supporting menu and drag and drop techniques.

Most existing editors fall somewhere between these two extremes. For example, several systems such as the Cornell Program Synthesizer [18], MacGnome [13], and Pecan [16] were mostly structured, but allowed users to enter a textual mode. When entered, the editor generated textual code from the structures, and users could then edit the text. When leaving the mode, the text was parsed, and users would have to correct any syntax errors before returning to the structured editing mode. Barista differs in that all of these interaction techniques are available in a single mode.

Several text-based editors, such as those in Eclipse and Visual Studio, and the editors from Harmonia, Ensemble and Pan [3], incrementally generate structure from the text when necessary, and use these structures to provide immediate feedback about errors, automatic formatting, and other tools. Barista takes the opposite approach by using structure as its primary representation and incrementally generating text when necessary.

The Proxima framework [17] is unique in that its editors can support edits on the visual structure of code. However, many of the edits can result in unexpected results. For example, in a Proxima editor, modifying a pretty-printed expression like that in Figure 9 requires the user to learn special rules for what will happen when backspace is typed at the boundaries of the view. Barista editors can overcome this by offering separate views of code for editing and reading, as in Figure 9 and Figure 10. Another recent hybrid editor, JPie [2], attempts to achieve flexibility by creating an edit-time grammar that adds support for "empty" expressions and "empty" operators. This is a more specific instance of Barista's support for adding error non-terminals and defining special classes of invalid tokens.

Flexibility of Presentation
We can also compare code editor implementation frameworks by the degree of control that they give designers to design the layout, appearance and structure of the visual presentation of code. Although these may seem like minor concerns, studies of programmers' maintenance task strategies suggests that users spend up to a third of their time just on the interaction techniques used to navigate and view code [10]. The study also suggested that helping programmers gather working sets of arbitrary fragments of code in a single workspace could remove many of these navigational bottlenecks; such views are difficult to implement with raw text, but would be easy with Barista because of its flexible presentation model.

Therefore, we believe that it is important that editors provide more structured views of code, enabling more direct manipulation, annotation and multiple views of individual structures in abstract syntax trees. A key design choice in Barista was to use a standard tree of *interactive* views like those found in any graphical user interface. Although many systems have used such trees of views to render code for viewing (for example, the use of HTML to represent JavaDocs), none of them have been fully interactive and editable. The use of interactive views also allows editor designers to take advantage of prior experience with GUI programming and they provide a hook for exploring the advantages of alternative sources of input that have been developed for regular GUIs, such as speech, gestures, and pens.

Other frameworks that provide a structured visual representation have only support a few conventional types of layouts. For example, the Proxima [17] and Ensemble [19] editors generally only use box and flow layouts to arrange text. Because Barista is based on a generic tree of interactive views, it can support these layouts and even free-form layouts, allowing editor designers to create editors in which users can organize code into visual groups.

Barista's presentation model, while allowing for more flexibility in the layout and appearance, does sacrifice the ability to have views whose structures differ substantially from the structure of their models. For example, Proxima [17] provides support for creating page layouts for word processors by creating extended versions of purely textual documents that add information about pages, headers and footers. Although this would be possible in Barista, it is not as well supported, because there is less correspondence between the models and the views. Nevertheless, this type of support is arguably less important for code editors.

Persistence of Structure
For many of the ideas that we illustrated that associate information with code (such as Figure 1 and Figure 12), the *persistence* of structure is an absolute requirement: modifications to code should preserve nearby structures and associated metadata whenever possible. For example, the method header information in Figure 1 should be preserved by the parser even if the left curly brace of the method declaration were removed, causing the parser to fail.

Commercial environments such as Eclipse do not preserve structure, making it difficult to associate information with code at all; instead, Eclipse generates structure from the text on demand and caches it. The Harmonia framework [3] makes many important contributions in ensuring persistent structure, by providing new kinds of parsers that can treat

parse trees as versioned documents. Barista does not currently have the same level of support for such parsers. One important concern that Harmonia does not seem to address, however, is warning users when a structure and its associated information will be lost. The token views used in Barista editors provide special notification mechanisms when a structure will be lost, so that the editor designer may warn the user before such operations are performed. Furthermore, Barista was designed to enable more usable structural edits (such as dragging statements and delimiters), which inherently preserve structure. Therefore, Barista can prevent such information losses by enabling the design of more usable, more efficient, and thus more enticing interaction techniques for editing and managing code structurally.

CONCLUSION

Relative to existing code editor implementation frameworks, Barista makes a number of contributions, providing editor designers with a standard, usable text-editing interaction technique and granting considerable freedom over the interactivity and visual representation of code and its associated information. We believe there is a bright future ahead for innovations in programming environments, and we hope that implementation frameworks like Barista will make such innovations more feasible, more usable, and more useful.

ACKNOWLEDGEMENTS

REFERENCES

1. Baecker, R. and Marcus, A., Design Principles for the Enhanced Presentation of Computer Program Source Text, *CHI* (1986), 51-58.

2. Birnbaum, B. E. and Goldman, K. J., Achieving Flexibility in Direct-Manipulation Programming Environments by Relaxing the Edit-Time Grammar, *VL/HCC* (2005), 259-266.

3. Boshernitsan, M., Harmonia: A Flexible Framework for Constructing Interactive Language-Based Programming Tools, *University of California, Berkeley*, Technical Report CSD-01-1149, 2001.

4. Garlan, D. B. and Miller, P. L., Gnome: An Introductory Programming Environment Based on a Family of Structure Editors, *SESPSDE* (1984), 65-72.

5. Graham, S. L., Harrison, M. A., and Munson, E. V., The Proteus Presentation System, *SESPSDE* (1992), 130-138.

6. Green, T. R. G. and Petre, M., Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework, *JVLC*, 7, (1996), 131-174.

7. Kehoe, C., Stasko, J., and Taylor, A., Rethinking the Evaluation of Algorithm Animations as Learning Aids: An Observational Study, *IJHCS*, 54, 2, (2001), 265-284.

8. Kelleher, C., Cosgrove, D., Culyba, D., Forlines, C., Pratt, J., and Pausch, R., Alice2: Programming without Syntax Errors, *UIST* (2002).

9. Ko, A. J., Aung, H., and Myers, B. A., Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing, *CHI* (2005), 1557-1560.

10. Ko, A. J., Aung, H., and Myers, B. A., Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks, *ICSE* (2005), 126-135.

11. Ko, A. J. and Myers, B. A., Citrus: A Language and Toolkit for Simplifying the Creation of Structured Editors for Code and Data, *UIST* (2005), 3-12.

12. LaToza, T., Venolia, G., and DeLine, R., Maintaining Mental Models: A Study of Developer Work Habits, *ICSE* (2005), to appear.

13. Miller, P., Pane, J., Meter, G., and Vorthmann, S., Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University, *ILE*, 4, 2, (1994), 140-158.

14. Oman, P. and Cook, C. R., Typographic Style Is More Than Cosmetic, *CACM*, 33, 1990, 506-520.

15. Pane, J. F., Myers, B. A., and Miller, L. B., Using HCI Techniques to Design a More Usable Programming System, *HCC* (2002), 198- 206.

16. Reiss, S. P., Graphical Program Development with Pecan Program Development Systems, *SESPSDE* (1984), 30-41.

17. Schrage, M. M., Proxima - a Presentation-Oriented Editor for Structured Documents, *Utrecht University* 2004.

18. Teitelbaum, T. and Reps, T., The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, *CACM*, 24, 9, (1981), 563-573.

19. Wagner, T. A. and Graham, S. L., Efficient and Flexible Incremental Parsing, *TOPLAS*, 20, 2, (1998), 980-1013.