Editable AI: Mixed Human-AI Authoring of Code Patterns

Kartik Chugh
Department of Computer Science
University of Virginia
Charlottesville, VA, USA
kc6afx@virginia.edu

Andrea Y. Solis
Department of Computer Science
George Mason University
Fairfax, VA, USA
asolis6@gmu.edu

Thomas D. LaToza
Department of Computer Science
George Mason University
Fairfax, VA, USA
tlatoza@gmu.edu

***Abstract*—Developers authoring HTML documents define elements following patterns which establish and reflect the visual structure of a document, such as making all images in a footer the same height by applying a class to each. To surface these patterns to developers and support developers in authoring consistent with these patterns, we propose a mixed human-AI technique for creating code patterns. Patterns are first learned from individual HTML documents through a decision tree, generating a representation which developers may view and edit. Code patterns are used to offer developers autocomplete suggestions, list examples, and flag violations. To evaluate our technique, we conducted a user study in which 24 participants wrote, edited, and corrected HTML documents. We found that our technique enabled developers to edit and correct documents more quickly and create, edit, and correct documents more successfully.**

***Index Terms*—Explainable AI, Autocomplete, Example-Centric Programming, Decision Trees, Development Environments**

I. Introduction

Developers authoring HTML documents define elements in ways which reflect patterns. For example, a developer might describe a navigation control by adding a number of button elements as children of a `<div>` tag, ensuring each button has a similar class which establishes its visual style and enables its association with logic in code. While elements in documents may be styled through Cascading Style Sheets (CSS), which describe visual properties which can be applied to elements, HTML documents exhibit patterns which reflect their structure. For example, a `div` element might contain only `link` elements, each with the same attribute. As developers work with HTML documents, developers often wish to make edits consistent with the existing structure. Developers might be supported in this activity through autocomplete suggestions, suggesting elements or attributes which reflect the document's structure, or might be informed when they write documents inconsistent with this structure. Offering this support requires a model of the code patterns which exist in the document.

Machine learning systems offer the possibility of identifying patterns from data. For example, a model trained on an HTML document might suggest that the most likely child element for a `div` container with a `class` value of `nav-bar` is an `img`. However, traditional machine learning systems lack explainability, offering no ability for the user to understand why the prediction was made. Moreover, in contexts where the patterns to be learned from data reflect patterns that the user themselves intended to create, the user may have a better model of expected behavior than the data itself. But traditional machine learning approaches lack editability, making it impossible for the user to correct or edit learned patterns to reflect their intent.

We envision a new form of human-computer collaboration in which the human (a developer) works together with the computer (the IDE) to author patterns reflecting a document's structure. A machine learning algorithm is first used to identify patterns. Using the model, the computer offers the developer code predictions, helping them complete tasks more quickly, and flags anomalies, helping them correct potential mistakes. When the code patterns learned from the data do not reflect the developer's true intent, the developer may view a representation of the computer's model, editing the model to reflect their intent.

We explore this approach in the context of a developer editing HTML documents. Patterns reflecting the structure of HTML elements and attributes are learned from individual HTML documents using a decision tree. As developers create new elements in the document, autocomplete suggests tag names and attributes based on the model. Developers may view the underlying model, viewing individual patterns identified (e.g., a parent tag of `<head>` and element tag of `<meta>` implies a `content` attribute). Developers may then see examples of each pattern, find code snippets which violate the pattern, and edit the pattern to better reflect their intent. We implemented this approach in a prototype tool called IRIS (Interactive Relationships Interface System), an extension to an HTML editor which enables developers to interact directly with code patterns.

To evaluate our approach, we conducted a user study in which 24 participants wrote, edited, and corrected HTML documents. We found that IRIS enabled participants to edit and correct documents more quickly and create, edit, and correct documents more successfully. Developers used IRIS to understand the source of autocomplete suggestions, developing trust in the system, as well as to identify examples of code patterns to better understand them.

978-1-7281-0810-0/19/$31.00 ©2019 IEEE

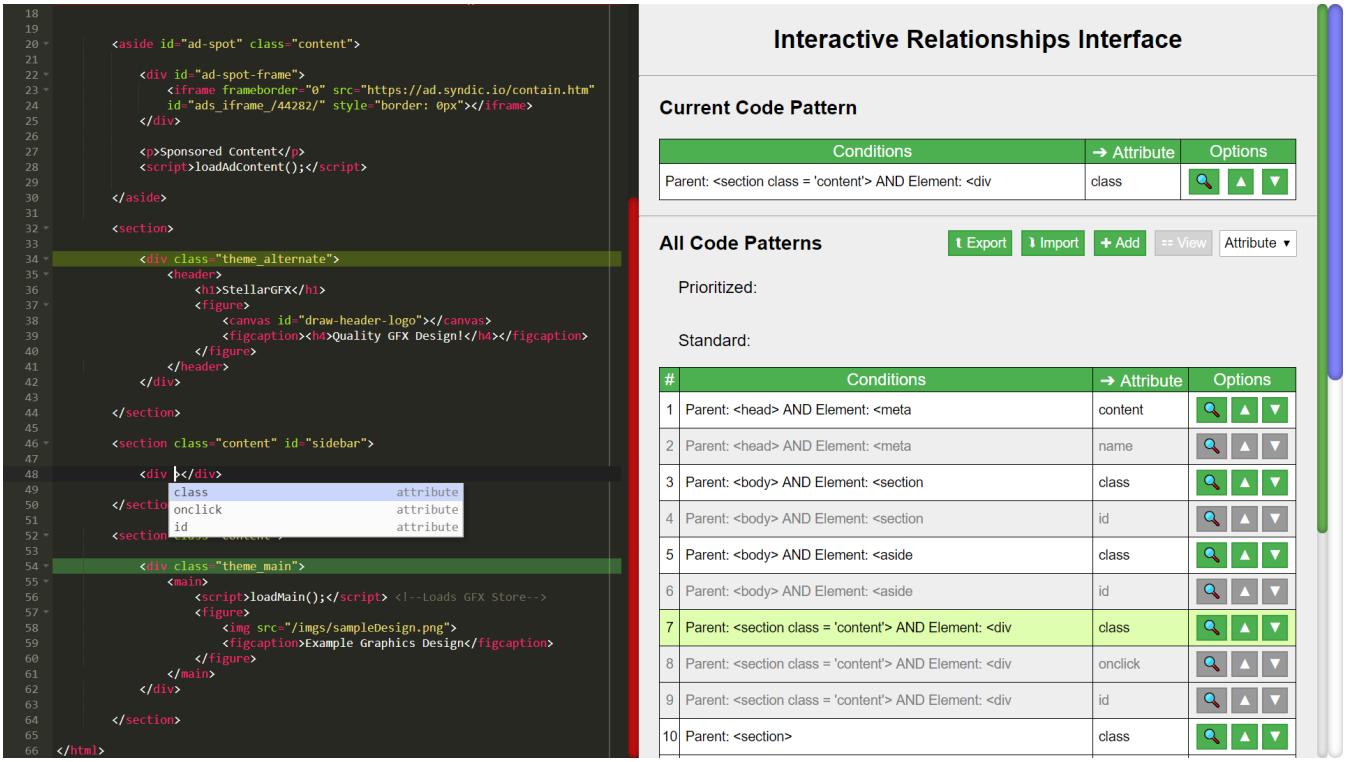

Fig. 1. As developers type in the HTML editor (left side), the autocomplete menu is displayed to offer suggested code completions. In the IRIS interface (right side), the Current Code Pattern explains to the developer the basis for the first suggested completion.

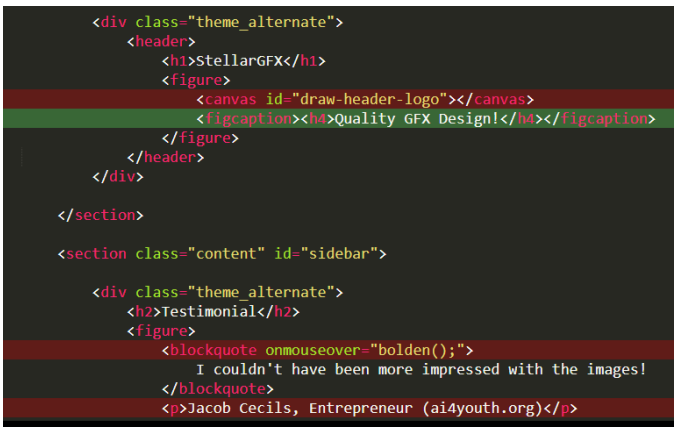

Fig. 2. Invoking the pattern inspector highlights examples of the code pattern (in green and yellow) and violations (in red) in the HTML editor. In this example, Alice is inspecting the pattern that `figure` parents contain `figcaption` children. Alice can see that the `figure` she is creating in the bottom 5 lines differs from the existing `figure` Bob created above, as hers uses a `<p>` tag in place of a `<figcaption>` tag.

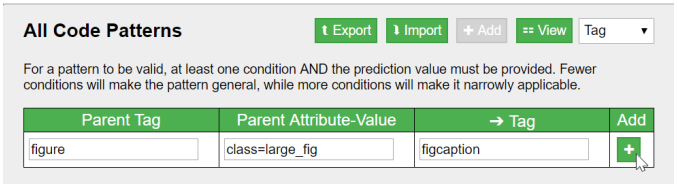

Fig. 3. Authoring a custom code pattern involves specifying a target feature and the condition feature(s) that imply it. In this example, Alice creates a code pattern that a `figure` with a `class` of `large_fig` should have a `figcaption` child element.

II. Motivating Example

Alice recently joined the web development team of a graphics design company. She has been tasked with completing the company's new webpage, which has been partially developed by her co-worker Barry. Alice loads his HTML document into an IRIS-augmented editor and scrolls down to the live preview panel to assess the current progress on the webpage. Observing its abundant white space and empty look, she decides to create a sidebar on the left side of the page.

As Alice types code which styles the sidebar, an autocomplete menu suggests several HTML attributes to apply, ordered by the system's confidence in each suggestion (see Fig. 1). Curious to understand why the first attribute was recommended to her, she looks at the Current Code Pattern panel, which describes the HTML features ("conditions") that guided the top attribute recommendation. In this case, Alice learns that a `<div>` tag—nested under a `<section class="content">` parent element—suggests that a `class` attribute follows. Alice clicks the magnifying glass icon, highlighting examples of the code pattern in the HTML editor (an activity we name "pattern inspection" in this paper). Reading a few of these examples, Alice realizes that Barry usually applied a `class` attribute

to elements like these, and sees examples of several of the classes he applied. With this insight, Alice mimics it in her sidebar, achieving a cohesive visual design.

After adding a client testimonial to the sidebar, Alice wants to verify that her new code follows the same general structure as Barry's. She selects Tag from the All Code Patterns dropdown and browses the list for patterns concerning related elements. For each such pattern, she uses the pattern inspection tool to see examples as well as violations in the HTML document. For example, in one code pattern Alice reads that a `figure` element contains a `figcaption` child element. The pattern inspector reveals that her `figure` instead uses a `p` element (highlighted in red), violating Barry's code pattern (see Fig. 2). Alice fixes her mistake by copying one of the `figcaption` pattern's usage examples (highlighted in green) and tweaking the caption to fit her image.

Having completed the sidebar, Alice intends to implement a footer displaying the logos of the company's partners. Alice decides that the `figcaption` pattern is overly broad and does not want it to apply to the logo images. To narrow its applicability, Alice downvotes the pattern, removing it from consideration, and clicks the Add button to write a more specific version: `<figcaption>` is the child tag of a `<figure class="large_fig">` parent element (see Fig. 3). She then adds this attribute-value pair to the captioned `<figure>` elements already in the document to comply with her new pattern. The system no longer recommends that Alice include `figcaption` inside unclassed `figures`, and will not flag her footer code for failing to do so. In this way, Fred, a future developer working in the same document to continue Alice's work, can be made aware of Alice's caption pattern.

Alice may also choose to share her code patterns with Michele. Michele is working on a different document, but wishes to use a similar look and feel. Alice can first use the Export button to download a JSON file containing the document's code patterns and send these to Michele. Michele can then click Import to import these code patterns into her document.

III. System

In IRIS, the computer and the human work together to create code patterns. The computer first learns code patterns from the HTML document by training decision trees. These decision trees are then used to offer the developer potential completions using an autocomplete interface, suggesting potential tags, attributes, or values. Developers may then interact directly with the code patterns, using a dedicated interface to view the code pattern responsible for a specific recommended completion, edit code patterns, and browse examples of code patterns. IRIS is implemented as an extension to a simple web-based HTML editor. In the following sections, we describe how IRIS learns code patterns from HTML documents, how code patterns are used to suggest code completions, and how developers may interact directly with code patterns.

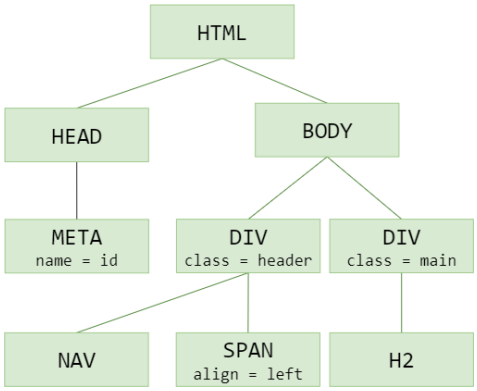

Fig. 4. IRIS parses HTML documents into an AST, where each node corresponds to a tag and may be associated with zero or more attributes and attribute values.

Tag	Parent Tag	Parent Attr/Val	Attribute
meta	head		name
div	body		class
span	div	class = header	align
div	body		class

Fig. 5. IRIS constructs training data tables for each of the decision trees for tags, attributes, and attribute values. The training data for attributes, shown here, includes the enclosing tag, parent tag, and its attribute-value pairs, as well as the resulting target attribute.

A. Learning Code Patterns

In this paper, we focus on HTML documents as individual tokens of HTML. *Tags*, *attributes*, and attribute values, or just *values*, are three important HTML tokens. Code patterns represent a relationship between between *condition features* and *target features*, where the former implies the latter.

To learn code patterns, IRIS first builds an abstract syntax tree (AST) for the active HTML document using Himalaya.js[1](shown in Fig. 4). As code patterns may or may not be document-specific, by default all code patterns are learned from and applied to an individual document.

IRIS constructs three separate decision trees corresponding to the three target types it may predict: an HTML tag, attribute, or value. IRIS bases its predictions on *features* extracted from the AST. In examining code patterns, we found that many patterns are contextual, reflecting the role of an element within the HTML document. For example, a tag contained in a `div` with a `class="sidebar"` attribute-value pair might vary from a tag contained in a `table`. Thus, the key condition features we chose to extract are the parent element tag and its attribute-value pairs. Additionally, for attribute and value targets, the features considered include the tag of the element that the developer is currently completing. When predicting values, the preceding attribute name of the current element is

[1]https://github.com/andrejewski/himalaya

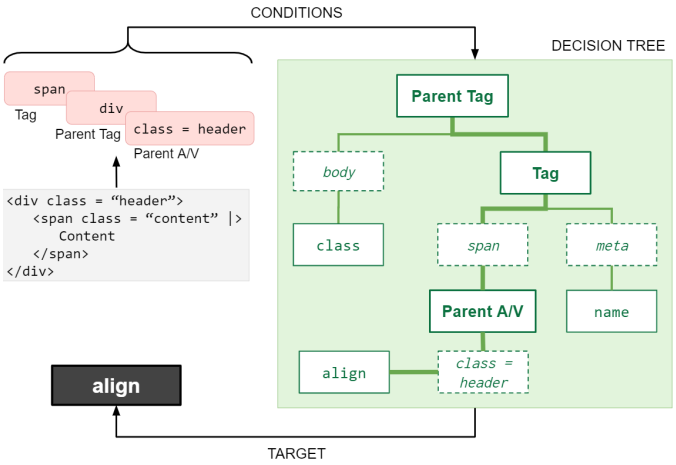

Fig. 6. Condition features are extracted from the current code context and used by the decision tree to predict targets. Only one target per decision node is shown for simplicity. The tree path taken in this example is bolded.

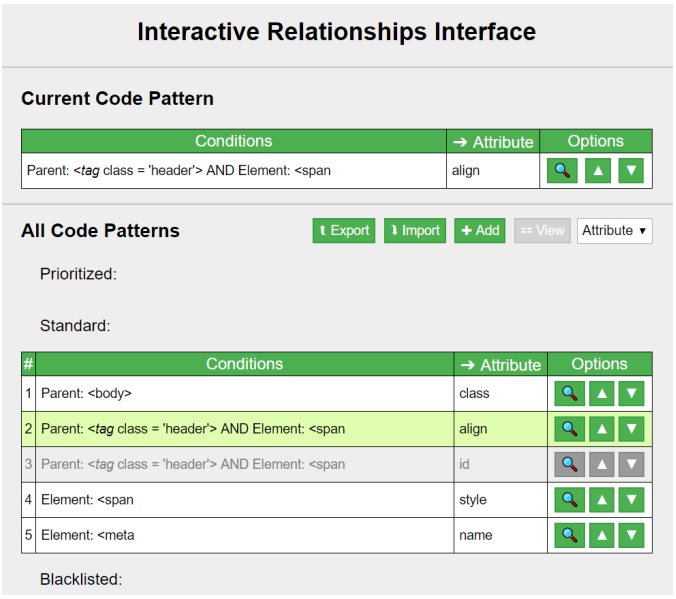

Fig. 7. The IRIS interface displays the code pattern for the current first listed autocomplete suggestion (if applicable) and tables of all code patterns.

considered. IRIS collects the relevant features from the AST, populating a table of training data (shown in Fig. 5) for the target type. After gathering the training data, IRIS constructs a decision tree for each target type. Using a JavaScript implementation[2] of the ID3 algorithm [1], a decision tree is learned from the training data.

B. Using Code Patterns to Suggest Completions

As the developer enters each character in the HTML editor, IRIS offers the developer autocomplete suggestions. To determine the type of target (tag, attribute, or attribute value) to be completed, IRIS first tokenizes the characters in the current line, from the first character to the cursor position. IRIS then uses the two preceding tokens to determine the type of the target. Consider the following example (`|` represents the cursor position and `whsp` indicates whitespace):

```
<span class="content" |>
```

is tokenized as:

```
(tag)(whsp)(attribute)(value)(whsp)
```

IRIS determines that a value followed by whitespace indicates that the next token may be an additional attribute. The target type is thus attribute.

After determining the target type, IRIS consults the decision tree for this target type to generate potential completions. To ensure that the decision tree reflects the current version of the code, IRIS lazily trains the appropriate decision tree on demand. IRIS then retrieves the information about the context of the current cursor position. This includes the parent tag, attributes, and attribute values as well as the current tag and attribute, if applicable. With this data, IRIS then uses the decision tree to generate possible completions (Fig. 6). These completions are displayed to the developer through an autocomplete interface (Fig. 1).

C. Interacting with Code Patterns

1) Viewing Code Patterns: To help developers understand code patterns and explain code completions, developers may directly view the set of code patterns. To offer developers a compact and understandable representation, we chose to present code patterns to the developer as a table of IF/THEN rules. Each rule describes a set of conditions in which it applies and the resulting predicted target. From each decision tree, each path through the decision tree, representing a series of conditions and a predicted target, is represented as a distinct code pattern. If multiple targets exist in a decision node, code patterns are constructed with identical conditions and the respective target. These patterns are stored in the target order expressed in the decision node, so as to preserve the ranking of more prevalent targets above less prevalent ones.

Code patterns are displayed to the developer in the IRIS interface to the right of the HTML editor (Fig. 7). When autocomplete is active, the Current Code Pattern at the top describes the code pattern associated with the top autocomplete recommendation. Below, tables for All Code Patterns display the code patterns for the document to the developer. The developer may toggle between viewing code patterns for tags, attributes, and attribute value targets using the dropdown in the upper-right of the All Code Patterns section. Each pattern entry lists the conditions under which the code pattern applies and the predicted target tag, attribute, or attribute value. The current code pattern is indicated in the All Code Patterns section with a light green background.

To enable developers to better understand the meaning and use of each code pattern, IRIS enables developers to see examples of each code pattern in the HTML editor. Clicking

[2]https://github.com/willkurt/ID3-Decision-Tree

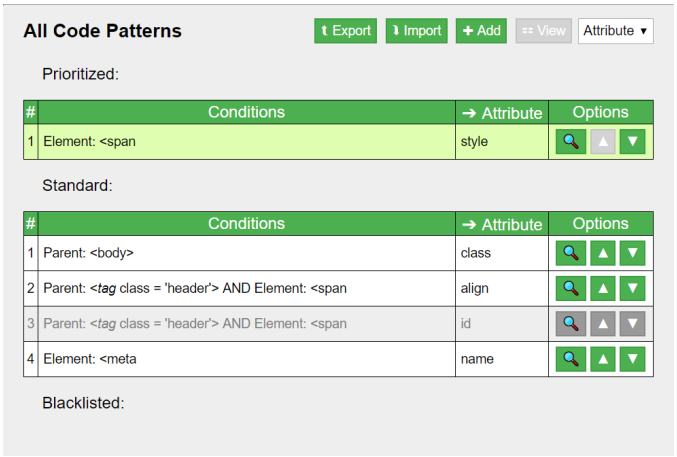

Fig. 8. Promoting a standard pattern moves it to the list of prioritized patterns.

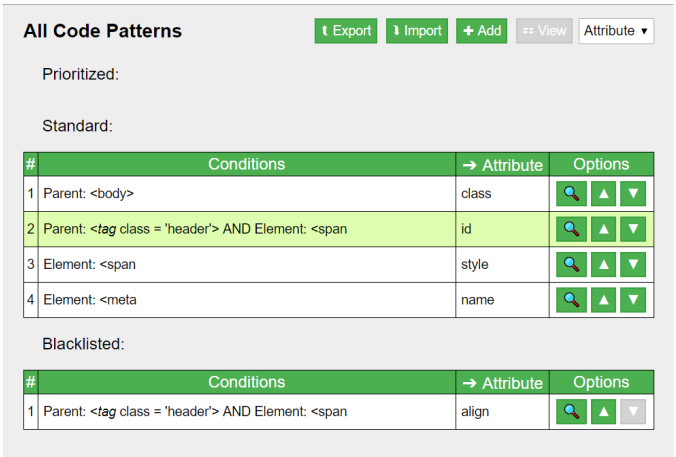

Fig. 10. Demoting a standard pattern moves it to the blacklist.

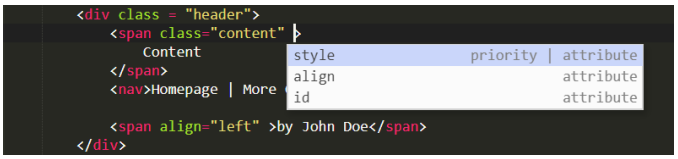

Fig. 9. Prioritizing patterns elevates their target features to the top of the autocomplete menu.

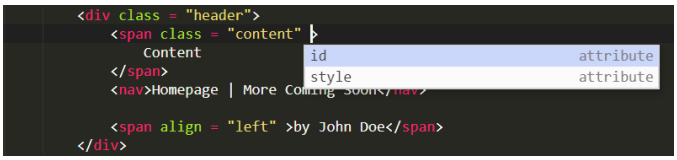

Fig. 11. Blacklisting patterns hides their target features from the autocomplete menu even if the pattern is applicable in the current context.

the magnifying glass next to the rule invokes the pattern inspector, highlighting examples of the code pattern. If the example is a positive example, where the conditions and prediction match exactly, it is highlighted in green. If the conditions are similar but unequal, the example is highlighted in yellow. Conditions are considered similar if they differ only in one or more of the parent element's attribute-value pairs. Developers are also shown examples which violate the rule, shown in red, indicating a potential defect in document structure, visual style, or both.

2) Editing Code Patterns: When inspecting code patterns, developers may identify patterns that they believe to be correct or incorrect or wish to manually create new code patterns expressing their intent. In these cases, the developer has insight into code patterns which they may offer to the computer. By taking the time to express their intent, developers may then receive suggested completions that better reflect the patterns they expect the document to follow.

In some cases there may be conflicting code patterns. Conflicting code patterns occur when there are multiple code patterns with identical conditions but which make different predictions. This reflects a HTML document which is inconsistent. In these cases, IRIS groups these code patterns together. The code pattern that is most likely is listed first and alternative code patterns are listed next and presented with a gray background and faded font. By grouping related code patterns, the developer may see alternatives and choose to indicate which they believe best reflects their intent.

Developers may indicate that they believe a code pattern to be correct by upvoting the pattern (up arrow icon) or to be incorrect by downvoting the pattern (down arrow icon). This may be indicated from both the current code pattern and the list of All Code Patterns. Upvoting or downvoting a code pattern toggles it between three states: standard, prioritized, and blacklisted. Developers may assign any priority to any code pattern at any time.

When first learned in training a decision tree, code patterns are initially in the *standard* state. Upvoting a standard pattern transitions it into the *prioritized* state. Prioritized patterns reflect insight that the developer themselves has offered into code patterns. Once in the prioritized state, code patterns are displayed in a separate section of the All Code Patterns panel, enabling developers to see prioritized patterns at a glance (shown in Fig. 8). Suggestions from prioritized patterns are always listed first in the autocomplete list (Fig. 9).

Standard patterns may be demoted to enter the *blacklisted* state. This state reflects the developers' indication that the code pattern does not match their intent. As such, blacklisted code patterns are never shown to the developer as a potential code completion. In addition, to prevent IRIS from learning other similar patterns, any entry in the training data that matches a blacklisted code pattern is removed. This enables the developer to give feedback on a single code pattern and for IRIS to incorporate this insight more broadly.

Because standard patterns are machine learned, the standard patterns list is continually updated as changes are made to the document. Standard code patterns may be automatically change or removed at any time. Prioritized or blacklisted code patterns which reflect the developers' intent remain until their priority is edited by the developer.

Developers may also choose to manually enter a prioritized code pattern by clicking the Add button in the All Code Patterns panel. The developer can first choose whether to create a tag, attribute, or attribute value code pattern and then create conditions and specify the target. Custom code patterns must have a target and at least one condition to be valid.

In cases where developers have taken the time to create prioritized or blacklisted code patterns, developers may wish to share these code patterns for use in other HTML documents. Developers may use these rules to capture a look and feel, which might be shared with other developers. Developers can click Export to download a JSON file containing prioritized and blacklisted patterns for the current document. Developers may then invoke Import to load code patterns into IRIS.

IV. Evaluation

A. Method

To investigate the impact of enabling developers to interact with code patterns, we conducted a user study. We recruited twenty-four participants through personal contacts and social media. All participants had prior experience in web development (average 3.2 years). 17 were male, and 7 female. Participants were paid $25 for 2 hours of their time.

Participants were assigned to a control or experimental condition. Participants in the control condition were provided a baseline HTML editor with autocomplete functionality, where code recommendations were generated through the same process as used in IRIS (Section III-B). However, the IRIS panel was disabled, and developers were unable to view or edit code patterns. Participants in the experimental condition were provided the same HTML editor augmented with a fully functional IRIS. Participants were each assigned to work on two tasks out of three possible tasks.

To encompass a range of potential tasks in which developers might benefit from IRIS, we designed three tasks: a *creation*, *continuation*, and *correction* task. This was intended to sample a range of situations in which developers might need to interact with code patterns. In the *creation* task, participants were directed to build an HTML code document from scratch in accordance with provided specifications. The specifications outlined document elements to create (e.g., *Create a two-column table in the center of the page*) and their expected styling (*Color the header the same as the footer*). In the *continuation* task, participants were directed to complete an unfinished HTML document in accordance with provided specifications. Participants were given a 400 line HTML document. The specifications asked participants to replicate specific elements (e.g., *Add a third link to the navigation bar*) as well as to restyle the document in specific ways (*Re-style the buttons in the second row to match those in the first*). In the *correction* task, participants were directed to find and fix inconsistencies in a document. These inconsistencies included missing or incorrect HTML features (e.g. a `<p>` tag on line 34 should be a `<caption>`) and relationships (`div` on line 252 should be nested under a `aside` parent).

TABLE I
Average task time by task and condition

	Mean task time	
Task	*Control*	*IRIS*
Creation	40.3	34.4
Continuation	57.1	44.1
Correction	44.1	29.4

TABLE II
Average task success by task and condition

	Mean success score		
Task	*Control*	*IRIS*	*Max. Possible Score*
Creation	11.4	14.9	17
Continuation	11.4	17.0	21
Correction	14.8	23.5	27

At the beginning of the study, participants were first asked to complete a brief tutorial explaining the main features of the HTML editor and, for experimental participants, IRIS. Participants were then given up to 75 minutes to complete each of the two main tasks. Participants were instructed to notify the researcher when they felt they had completed a task. As participants worked, we collected a screen recording for analysis. After completing the tasks, we interviewed participants about their experiences.

B. Results

1) Creation Task: Participants using IRIS finished building the outlined webpage after an average of 40.3 minutes, compared to 34.4 for control participants. However, the Welch's one-tailed t-test revealed this difference only approached significance ($p = 0.06$). To evaluate participants' success, we scored each HTML document created, awarding one point for each item of the task specification they successfully completed. Participants with IRIS successfully completed significantly more requirements ($p < 0.01$), completing an average of 14.9 compared to 11.4 by control participants.

Several experimental participants opted to focus on viewing code patterns, rather than editing code patterns. These participants periodically browsed the All Code Patterns list to assess their progress. As one participant explained, *"I didn't really need to highlight samples of code I just wrote. But just having the list there helped me keep track of what work I've already done and what I have left."* Others noted that seeing the existing patterns in their code was useful for both evaluating task-compliance and conceiving ideas for what to develop next.

Other experimental participants used IRIS to manually define their own code patterns. One participant explained that adding their own patterns ahead of time helped with *"sticking to a plan"*, while another observed that it *"made the autocompletes [sic] more useful"* by transferring his intent to the system. These participants made extensive use of promoting

and demoting code patterns, enabling them to remove patterns which overfit the data and focus on intended patterns.

2) Continuation Task: Participants with IRIS completed the continuation task significantly faster ($p = 0.03$), finishing in 57.1 minutes compared to 44.1 minutes for control participants. To score participants' success, we gave participants one point for each specification they satisfied and one or two points for following the specification consistently throughout their document. This led to a maximum score of 21 points. Experimental participants performed significantly better ($p < 0.01$), with an average progress of 17.0 rather than 11.4.

IRIS enabled participants to more quickly identify and reapply code patterns within their document. Participants with IRIS often relied on autocomplete recommendations to develop code immediately and later, briefly review the Current Code Pattern and highlight usage examples to double check the recommendations' applicability. Compared to control participants with suggested completions but no insight into their source, IRIS participants developed stronger trust in the recommendations and relied on them more heavily over time. In contrast, several control group participants appeared skeptical of the recommendations, either ignoring them or spending substantial time searching for examples to verify them. A few IRIS participants used upvoting or downvoting to edit code patterns. One participant observed that *"The autocomplete got even more accurate as I voted on the patterns"*, making document development *"easier and easier"*.

IRIS participants made use of the code patterns list to learn code patterns, locating them by highlighting usage examples and reapplying them in new code. Participants used the patterns list to understand the appropriate attribute value to use in various contexts, enabling them to reproduce appropriate attribute-value pairs for a given element. Many participants browsed the code patterns for its parent tag conditions, which aided them in reproducing appropriate parent-child structures. For example, participants first became aware of the rule to nest self-contained `img` elements inside a `figure` parent by seeing this code pattern in the list (as a new semantic element introduced in HTML5, the `<figure>` tag and its usage may not be widely understood). Most participants followed up by highlighting examples of this code pattern, either recreating or copying document snippets involving `figure` and `img`. In contrast, control participants often incorrectly created `img` elements without `figure` parents, and more generally, were not as aware of the existence of patterns.

3) Correction Task: IRIS participants completed the correction task significantly faster ($p < 0.01$), finishing in an average of 29.4 minutes compared to 44.1 minutes for control participants. To score participants' success, we gave participants one point for the addition, modification, or deletion of code to make it consistent with the code patterns in the document. This led to a maximum score of 27 points. IRIS participants were significantly more successful ($p < 0.01$), with an average score of 23.5 compared to 14.8 for control participants.

A key benefit IRIS offered participants was the ability to highlight pattern examples and violations. All participants with IRIS used the pattern inspector to scan the HTML document for inconsistencies. Participants generally interpreted the red highlights to indicate a defective code feature, and the yellow highlights to suggest a missing or defective attribute-value pair. This heuristic was often helpful for participants in identifying inconsistent code. However, it sometimes misled a few participants into "fixing" elements that did not need correction, as certain defects featured predominantly in the document, and thus, were listed by IRIS as code patterns. Despite this occasional shortcoming, the patterns list and inspection tool were instrumental in helping participants successfully find and repair code defects:

> *"The magnifying glass [button] was very handy. For each pattern I pressed [the button], read the colored lines and compared them... And then figured out which line needed fixing from there."* (IRIS Participant)

In contrast, control participants struggled to locate inconsistencies, particularly those concerning HTML values.

> *"Sorting through all the code was really confusing and time-consuming. I couldn't tell what I was supposed to do for the most part... Only a few blatantly wrong tags stood out to me."* (Control Participant)

Related Work

Our work builds on prior work in autocomplete tools for developers, systems which identify and make use of patterns in code, and work in explainable AI.

Autocomplete is one of the most widely used features in modern development environments, with one study finding that 6.7% of all of developers' IDE commands were code completions [2]. A number of systems have explored techniques for offering more effective autocomplete interactions. Many of these systems build a model of document *context* to offer more accurate predictions. CSCC offers developers potential method calls as code completions based on the context of surrounding method calls [3]. Dompletion builds a model of an HTML document object model, enabling suggestions of valid completions which respect element types [4]. Calcite crowdsources the creation of virtual methods, enabling developers to use code completion for methods that developers expect to exist but do not [5]. Proksch et al. discuss some of the challenges in evaluating method recommender systems [6]. Jungloid mining enables developers to find sequences of method calls which convert from an object in one type to an object of a different type [7]. Work has explored using code history information to improve code completion [8]. Other tools have explored approaches for completing entire method bodies from prior examples or code clones [9]. Active code completion offers the developer palettes for creating specific complex literals [10]. Beyond code completion, other approaches help connect code more effectively to documentation to help developers more effectively find methods. For example, work has explored using documentation to augment interactions in the IDE [11].

A variety of autocomplete systems have built statistical models of documents, using these models to power better

autocomplete. These systems largely rely on the inherent high predictability and repetitiveness of code [12]. Systems such as MAPO [13], GraPacc [14], and method call relationship graphs [15] offer approaches for capturing patterns in API method usages. Other work has explored approaches for learning method completions from code repositories [16], [17]. Recent systems have expanded the amount of interactivity, enabling developers to interactively enter keywords and choose among alternatives to refine completions [18].

Building on the growing fear of AI systems creating a "black box society" [19], work in the area of explainable AI has explored ways in which difficult to understand machine learning models may be represented and communicated more simply to users [20]. For example, model understanding through space explanations enables translating an arbitrary black-box representation of a machine learning system into decision sets which capture the behavior of the black box in specific circumstances [21]. In interactive machine learning, users can view and correct classifications made by a system [22]. Techniques such as why-oriented [23] and explanatory debugging [24] enable users to view predictions made by a naive Bayes model and make corrections back to the model. Our work builds on these ideas, applying the idea of interactive machine learning to document editing through code completion and viewing violations as well as exploring the use of rules as a means to view and edit machine learning models.

V. Limitations and Threats to Validity

Our results might vary for larger HTML documents or for documents which have been created over a longer period of time, where the number of code patterns is larger. While the potential value of our approach in managing more patterns may grow, there may also be a larger potential for finding spurious or overlapping patterns. In our study, we observed developers creating a document they began themselves, working to stay consistent, as well as modifying an existing document that they had not written. Our results might vary for developers working in the same document over a period of time who have internalized more code patterns.

VI. Discussion

In this paper, we proposed an approach for *editable* AI, in which the human and computer work together to create and maintain code patterns. Code patterns are first learned by the computer through decision trees. Code patterns are then used to power code completions, suggesting ways in which developers may write code which follows code patterns. To help developers explain code patterns, developers can view a compact representation of code patterns, comparing alternative predictions and viewing examples to see the source of predictions. Developers may use patterns to identify violations, viewing examples in the document which violate patterns. In cases where learned code patterns overfit the document and do not reflect developers' true intent, developers may downvote code patterns, which the system can then use to prune training data to prevent similar rules from being learned. Or developers may signal that a code pattern matches their intent by upvoting or authoring their own code patterns.

Our results reveal that editablity and explainability matter. Compared to developers offered autocomplete with the same learned patterns, developers with the ability to view and edit code patterns through IRIS were able to edit and correct HTML documents more quickly and create, edit, and correct HTML documents more successfully. Developers were able to use code patterns to explain suggested completions, gathering examples to see the source of these recommendations and increasing their trust in the system. Developers used code patterns to identify and correct violations. As developers developed their own intent, developers reflected this by editing code patterns, enabling autocomplete to offer better recommendations. In this way, editability and explainability helped developers to author more consistent and well-structured documents. This suggests the potential benefit, for creative work, of offering the user greater control over machine learning.

Code patterns offer a way of capturing the look and feel of a document. By representing the structure of the elements which exist in an HTML document, developers can more directly understand and interact with the visual look and feel. For example, code patterns might capture that `img` elements in the header should each have a particular size as expressed by a `class`. Our representation of code patterns is limited in a number of respects. In representing code patterns, we included features such as parent tags and attributes to capture the context-specific nature of HTML document structure. But additional contextual features likely sometimes matter, from the numeric order of an element to the characteristics of other ancestor elements. More expressive representations might capture these. Code patterns are complimentary to CSS, helping capture document structure rather than only visual styling. Compared to CSS styles, code patterns are more flexible, as unlike CSS styles which automatically apply to elements, developers are free to create elements whether or not they follow a code pattern. However, there may be cases where code patterns and CSS styles more directly overlap, and it may sometimes be helpful to enable code patterns to be converted to CSS rules or vice versa.

Our approach might also be applied to other documents where the user has intent in mind about its structure. Most directly, our approach might be applied to code written in a programming language. However, such patterns may be considerably more complex, involving a much wider variety of condition features and targets. While existing statistical approaches to modeling code may offer hints into appropriate representations, more work remains to understand appropriate ways of learning these patterns and how they might be succinctly communicated to developers.

Acknowledgments

We thank our study participants for their time. This work was conducted in part while Kartik Chugh was an intern in the Aspiring Scientists Summer Internship Program at George Mason University. Andrea Solis was supported in part by an

Undergraduate Research in Educational Data Mining grant, NSF IIS-1757064.

REFERENCES

[1] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, Mar. 1986.

[2] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse ide?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, July 2006.

[3] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "Context-sensitive code completion tool for better api usability," in *International Conference on Software Maintenance and Evolution*, 2014, pp. 621–624.

[4] K. Bajaj, K. Pattabiraman, and A. Mesbah, "Dompletion: Dom-aware javascript code completion," in *International Conference on Automated Software Engineering*, 2014, pp. 43–54.

[5] M. Mooty, A. Faulring, J. Stylos, and B. A. Myers, "Calcite: Completing code completion for constructors using crowds," in *Symposium on Visual Languages and Human-Centric Computing*, 2010, pp. 15–22.

[6] S. Proksch, S. Amann, S. Nadi, and M. Mezini, "Evaluating the evaluations of code recommender systems: A reality check," in *International Conference on Automated Software Engineering*, 2016, pp. 111–121.

[7] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: Helping to navigate the api jungle," in *Conference on Programming Language Design and Implementation*, 2005, pp. 48–61.

[8] R. Robbes and M. Lanza, "How program history can improve code completion," in *International Conference on Automated Software Engineering*, 2008, pp. 317–326.

[9] R. Hill and J. Rideout, "Automatic method completion," in *International Conference on Automated Software Engineering*, 2004, pp. 228–235.

[10] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers, "Active code completion," in *International Conference on Software Engineering*, 2012, pp. 859–869.

[11] M. Goldman and R. C. Miller, "Codetrail: Connecting source code and web resources," in *Symposium on Visual Languages and Human-Centric Computing*, 2008, pp. 65–72.

[12] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Commun. ACM*, vol. 59, no. 5, pp. 122–131, Apr. 2016.

[13] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *European Conference on Object-Oriented Programming*, 2009, pp. 318–343.

[14] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Grapacc: A graph-based pattern-oriented, context-sensitive code completion tool," in *International Conference on Software Engineering*, 2012, pp. 1407–1410.

[15] X. Li, Z. Wang, Q. Wang, S. Yan, T. Xie, and H. Mei, "Relationship-aware code search for javascript frameworks," in *International Symposium on Foundations of Software Engineering*, 2016, pp. 690–701.

[16] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *European Software Engineering Conference and the Symposium on The Foundations of Software Engineering*, 2009, pp. 213–222.

[17] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Conference on Programming Language Design and Implementation*, 2014, pp. 419–428.

[18] X. Rong, S. Yan, S. Oney, M. Dontcheva, and E. Adar, "Codemend: Assisting interactive programming with bimodal embedding," in *Symposium on User Interface Software and Technology*, 2016, pp. 247–258.

[19] F. Pasquale, *The Black Box Society: The Secret Algorithms That Control Money and Information*. Cambridge, MA, USA: Harvard University Press, 2015.

[20] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi, "A survey of methods for explaining black box models," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 93:1–93:42, Aug. 2018.

[21] H. Lakkaraju, "Faithful and customizable explanations of black box models," in *AAAI/ACM Conference on Artificial Intelligence, Ethics, and Society*, 2019.

[22] J. A. Fails and D. R. Olsen, Jr., "Interactive machine learning," in *International Conference on Intelligent User Interfaces*, 2003, pp. 39–45.

[23] T. Kulesza, S. Stumpf, W.-K. Wong, M. M. Burnett, S. Perona, A. Ko, and I. Oberst, "Why-oriented end-user debugging of naive bayes text classification," *ACM Trans. Interact. Intell. Syst.*, vol. 1, no. 1, pp. 2:1–2:31, Oct. 2011.

[24] T. Kulesza, M. Burnett, W.-K. Wong, and S. Stumpf, "Principles of explanatory debugging to personalize interactive machine learning," in *International Conference on Intelligent User Interfaces*, 2015, pp. 126–137.