Towards More Effective AI-Assisted Programming: A Systematic Design Exploration to Improve Visual Studio IntelliCode's User Experience

Priyan Vaithilingam*, Elena L. Glassman*, Peter Groenewegen†, Sumit Gulwani†,
Austin Z. Henley†, Rohan Malpani†, David Pugh†, Arjun Radhakrishna†, Gustavo Soares†,
Joey Wang†, Aaron Yim†
*Harvard University
†Microsoft

***Abstract*—AI-driven code editor extensions such as Visual Studio IntelliCode and Github CoPilot have become extremely popular. These tools recommend inserting chunks of code, with the lines to be inserted presented *inline* at the current cursor location as *gray text*. In contrast to their popularity, other AI-driven code recommendation tools that suggest *code changes* (as opposed to code completions) have remained woefully underused. We conducted lab studies at Microsoft to understand this disparity and found one major cause: *discoverability*. Code change suggestions are hard to surface through bold, inline interfaces and hence, developers often do not even notice them.**

Towards a systematic understanding of code change interfaces, we performed a thorough design exploration for various categories of code changes: *additive single-line changes*, *single-line changes*, and *multi-line changes*. Overall, we explored 19 designs through a series of 7 laboratory studies involving 61 programmers and distilled our findings into a set of 5 design principles. To validate our results, we built and deployed a new version of IntelliCode with two of our new inline interfaces in Microsoft Visual Studio 2022 and found that they lead to a significant increase in usage of the corresponding tools.

***Index Terms*—inline-suggestion, AI-suggestion, refactoring, iterative-refinement, code-completion**

I. Introduction

Integrated Development Environments (IDEs), such as Visual Studio [1], provide tool support for several code editing tasks, from writing new code to modifying existing ones. Recently, AI-driven code editor extensions such as IntelliCode [2] and Copilot [3] introduced new features that can recommend entire lines of code. These features adopted the *gray text* interface to preview the suggestion *inline* in the editor as shown in Figure 1. They quickly became popular, and IntelliCode was ranked best feature in Visual Studio 2022 [4] and Copilot is now widely studied in academia [5]–[7].

AI-driven extensions are also starting to help programmers to modify existing code. IntelliCode uses program synthesis techniques [8], [9] to automate custom repetitive code changes. It leverages the past changes performed by the developer in the IDE to predict the next code changes they will do. Figure 2 shows an IntelliCode suggestion to automate a custom repetitive change that a developer was manually performing. The suggestion is shown using the *lightbulb* interface (Figure 2a), which is traditionally used in IDEs to present code change suggestions. The developer has to manually click the *lightbulb* and read through the menu items, to see all available code change suggestions.

```
[HttpPost]
[AllowAnonymous]
0 references | 0 changes | 0 authors, 0 changes
public IActionResult Upload(FileViewModel fileViewModel)
{
    if (fileViewModel == null)   Tab  to accept
}
```

Figure 1: An inline *gray text* suggestion produced by Visual Studio's IntelliCode feature.

In contrast to the gray text experience, the Visual Studio IntelliCode telemetry showed that code changes suggested through lightbulb interfaces are heavily underused. We conducted lab studies at Microsoft to understand why developers were not using this feature in IntelliCode and we identified one major issue: *discoverability*. Unlike inline *gray-text* interface in which suggestions are presented *proactively* to the developer, lightbulbs require manual initiation. However, the developer is not always aware of all the suggestions available via the lightbulb interface. Sometimes, they don't even realize that a suggestion is being presented via a lightbulb. This problem is aggravated with AI-assisted suggestions, because these suggestions are significantly more dynamic with their availability changing frequently due to the constant improvements to the AI model. This observation corroborates with past studies that identified several usability issues that lead to the underuse of refactoring tools in IDEs [10]–[12].

The limitations of existing interfaces for suggesting code changes and the success of the inline *gray text* interface for AI-assisted code insertions, inspired a natural question: *How can we leverage inline interfaces to effectively show AI-assisted code changes?*

We performed a systematic design exploration to answer the above question. In particular, we investigated a variety of

```
39
40        NotNullOrEmpty((y) => y.Description);
41
42        RuleFor(x => x.TagList).NotNull().NotEmpty();
43      }
44    }
```

(a) An ellipses is shown under `RuleFor` indicating suggestions are available. A lightbulb is displayed on navigating to the ellipses.

```
42        RuleFor(x => x.TagList).NotNull().NotEmpty();
   Use discard '_'
   Introduce local for 'RuleFor(x => x.TagList).NotNull().NotEmpty()'
   Wrap call chain
   Wrap and align call chain
   ★ IntelliCode suggestion based on recent edits: NotNullOrEmpty(x => x.TagList) ▸   Apply suggestion   Ctrl+Alt+.
   Suppress or Configure issues ▸                                                    Ignore suggestions like this

Lines 41 to 43
RuleFor(x => x.TagList).NotNull().NotEmpty();
NotNullOrEmpty(x => x.TagList);
}

51      }
52
        1 reference | 0 changes | 0 authors, 0 changes
53      public class CommandValidator : BaseValidator<Command>
```

(b) A full *diff-view* of the suggested changes on selecting the item on lightbulb menu.

Figure 2: An AI-assisted code change suggestions produced by IntelliCode after the developer had perform a similar change twice. The suggestion is progressively revealed through the lightbulb interface in Visual Studio 2022.

inline code change interfaces by testing 19 different designs that we explored across 7 user studies with 61 participants. Our method involved a user-centered design approach where we iteratively established hypotheses, created a set of design prototypes, ran user studies to elicit feedback on the designs, and performed open coding to gain insights from users.

Our designs addressed 3 separate categories of code change suggestions: (a) *Additive changes*, that suggest adding code tokens at multiple places beyond just at the current cursor location (see Figure 4); (b) *Single-line changes*, that suggest both deleting existing tokens and adding new tokens on a single line (see Figure 5); and (c) *Multi-line changes*, that suggest deleting and adding new code on multiple lines (see Figure 6). These categories require increasingly visually complex designs. At the end of our investigation, we ended up with the designs depicted in Figure 4 for additive changes, Figure 5(a) for single-line changes, and Figures 6(a) and 7(a) for multi-line changes.

Equally important, we also distilled 5 design principles including the need to juxtapose the original and modified code, reuse existing interfaces, and provide suggestions proactively. Note how each of the above designs follows the above principles, calling back to familiar software engineering interfaces of colored diff views and gray text and allow the developer to understand both the original and modified code at a glance. We validated the findings for two notable interfaces by implementing and deploying them as part of IntelliCode in Visual Studio 2022. These interfaces were used by several thousands of developers and led to a 3.5x overall increase in the usage of AI-assisted code changes produced by this tool.

Based on this work, we make the following key contributions:

- A systematic design exploration of 19 user interface designs for code change suggestions in a popular code editor;

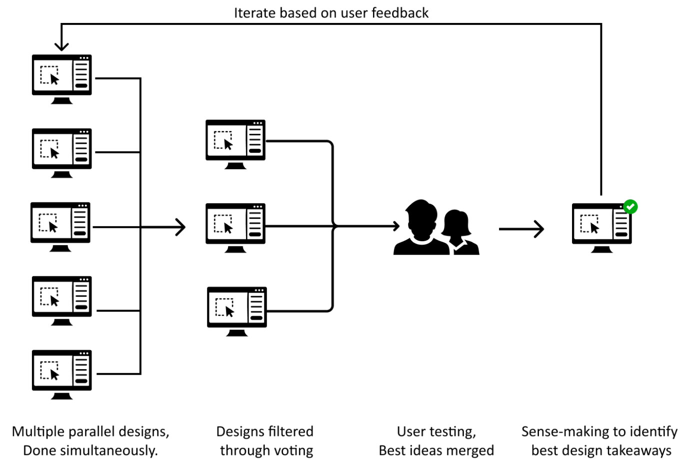

Figure 3: Design exploration and user testing strategy.

- Findings from a series of user studies on the designs involving 61 professional programmers;
- Findings from a large-scale field deployment of two notable prototype designs;
- Empirically grounded principles for the design of code suggestions based on the findings.

II. Methodology

To better understand how to leverage in-line interfaces for AI-assisted code changes, we (1) performed a systematic design exploration, (2) conducted a series of user studies, and (3) performed a field deployment of two notable designs.

A. Design Process

We use a combination of iterative design and parallel design techniques proposed in [13] (Figure. 3). Each design iteration contains three steps:

1) ***Parallel Design*** For each iteration, we come up with a set of hypotheses, and create 8-10 designs for the given set

Iterative Design Exploration			
Stage	**Iteration**	**Total designs explored**	**Participant count**
1. Additive code changes	1	1	12 ($P_i1 - P_i12$)
2. Single-line code changes	1	4	8 ($P_e1 - P_e8$)
	2	4	8 ($P_e9 - P_e16$)
3. Multi-line code changes	1	4	8 ($P_m1 - P_m8$)
	2	1	6 ($P_m9 - P_m14$)
	3	3	11 (P_m15 - P_m25)
	4	2	8 (P_m26 - P_m33)
Total	**7**	**19**	**61**

Table I: Design exploration table.

of hypotheses. Through multiple rounds of discussion and voting among the authors, we select up to 4 designs for user testing. Overall we created ∼50 designs, of which 19 designs were selected through voting to be user tested.

2) ***User testing*** We create prototypes using the selected designs from the previous step and run user studies to understand the usability of the designs. For each iteration, we typically run the user study with 8 participants.
3) ***Sensemaking*** Three of the authors do an open coding of participants' think aloud transcripts, informal interview responses, and observation notes for every iteration, and extract themes and learning that will be used for the next iteration or to create the final design.

Additionally, we have split the design exploration into three stages as shown in Table I. Each stage addressed a broad type of code change ranging from simpler code changes to more complex ones.

B. Study Procedure

We recruited 61 participants (59 Male, 2 Female) for user studies over 7 iterations. All the participants were Visual Studio users who have been using the IDE for at least 3 years. Each session was conducted remotely via Microsoft Teams with the environment pre-configured with out Visual Studio plugin, with each session taking around 45 minutes to complete. All sessions were audio and video recorded, including the participants' screens. We followed the same procedure within each iteration. We start with an informal interview about the participant's general code editing and refactoring experience with Visual Studio. This interview is followed by a brief overview of the code repository that the participant will be using during the study along with the tasks. Participants could ask questions about the code and tasks at any time. We also asked the participants to think out loud. We do not inform the participants about the existence of the inline suggestion tool. We then ask each participant to perform one task per prototype we are testing in the iteration (within-subjects study). Finally, we then conduct an informal interview to understand the participant's feedback. During the user study, we explore the following research questions.

1) **RQ1**: Do participants notice the suggestions?
2) **RQ2**: Do participants understand the proposed suggestions, i.e., which code the system suggests removing and what code it suggests adding?

```
8   class Obstacle
9   {
10      public ObstacleType type { get; set; }
11      public int XPos { get; set; }
12      public int YPos { get; set; }
13      public double XVelocity { get; set; }
14      public double YVelocity { get; set; }
15                                                        Tab accept | ...
16      public Obstacle (ObstacleType type, int xPos, int yPos, double xVelocity, double yVelocity)
        {
            Type = type;
            XPos = xPos;
            XVelocity = xVelocity;
            YVelocity = yVelocity;
        }
17  }
```
(a)

```
8   class Obstacle
9   {
10      public ObstacleType type { get; set; }
11      public int XPos { get; set; }
12      public int YPos { get; set; }
13      public double XVelocity { get; set; }
14      public double YVelocity { get; set; }
15      private bool IsValid { get; set; }                Tab accept | ...
16
17      public Obstacle (ObstacleType type, int xPos, int yPos, double xVelocity, double yVelocity, bool isValid)
18      {
19          Type = type;
20          XPos = xPos;
21          XVelocity = xVelocity;
22          YVelocity = yVelocity;
            IsValid = isValid;
23      }
24  }
```
(b)

Figure 4: Designs for Code Insertion using *gray-text* design.

3) **RQ3**: How much effort does it take to evaluate and act on the suggested changes?

After each iteration, the authors of the paper collate participant's think aloud audio transcripts, informal interview responses, and the observation notes for the sensemaking where we perform an open coding of the collected data over 2-3 sessions.

C. Implementation

For the parallel design step, we used the Figma[1] tool to create high-fidelity mock-ups. To test the selected designs with users, we used a combination of real prototypes built on top of Visual Studio IntelliCode and *wizard of oz* [14] low-cost prototypes built with Figma. We chose between building the prototype or using Figma depending on the complexity of the interface. Finally, for the field deployment, we developed production-level features in IntelliCode with the selected designs [15].

III. Design Exploration

A. Stage I: Designing for additive code changes

For the first stage of the design exploration, we started with a simple scenario—additive code changes. These changes add new code at, before, or after a particular location but do not remove or modify the existing tokens. For instance, Figure 4 shows two additive code changes. The first fills in the constructor parameters and assignments in an empty constructor. The second, adds a new parameter, and its corresponding assignment to a constructor. Note that in both examples, the added code is not continuous. Instead, the change places the new code in between existing content.

We consider these changes simpler to display using an inline interface because they do not modify any existing token, and thus, we can reuse the gray-text interface, which so far was used only for displaying suggestions that insert continuous sequence of tokens such as the one in Figure 1. Therefore,

[1]https://www.figma.com/

we start our design exploration with user testing *gray-text* interface for additive code change suggestions.

1) Experiment: Recently, we introduced a new feature in IntelliCode to detect when the developer is manually performing a change that is already implemented as a Visual Studio refactoring and surface the change to the developer. The underlining program synthesis technology that enables this feature is described in [16]. We selected two additive suggestions produced by this feature to test the gray text experience: *Complete constructor* and *Add parameters to constructor* (Figure 4). We performed just one iteration of design exploration with a prototype containing the inline *gray-text* design. Note that since these changes are also available through the default set of refactorings in Visual Studio and the participant could also choose to use the lightbulb if they were aware of this option. We conducted a user study with 12 participants ($P_i1 - P_i12$)[2]. Each participant had to perform two tasks: (1) Add a constructor to a given class, and initialize all the properties of the constructor by using parameters passed to the constructor and (2) Add two new properties to the class, and edit the existing constructor to initialize the newly added properties. Both tasks can be performed by existing VS tools by using the lightbulb menu.

2) Results: Nine of the 12 participants used the inline suggestions (experiment) to finish both the tasks. Two participants (P_i7 and P_i8) did not notice the inline suggestions as they were looking at the keyboard as they typed. P_i12 used the lightbulb icon (baseline) —the only participant to do so. This shows that the inline suggestions are more discoverable than suggestions in *lightbulb* icon. During the informal interview after the tasks, all of the 12 participants mentioned they preferred using the inline suggestions (for the three participants who missed seeing the suggestion while performing the tasks, we showed a preview). For instance, P_i3 said "*I like these suggestions. I don't even have to code. I just have to press TAB. That's basically all I want to do.*" During this informal interview, all of the 9 participants who used the inline suggestions were able to explain the proposed suggestion to us for both the continuous and discontinuous scenarios. One participant (P_i5) also drew a parallel with their prior experience with a tool that uses inline suggestion by saying, "*This feels like Copilot, I have tried the beta version for some personal projects. I like this—it reminds me of Copilot.*" Seven participants explicitly showed approval and delight when performing the task. However one participant (P_i10) expressed concerns over the usefulness of such suggestions for the given scenario: "*I'm not entirely sure if that is a huge time saver though. Maybe just a few seconds. It's nice.*" Further, three participants (P_i2, P_i9, P_i10) mentioned they will not be comfortable accepting the suggestion if the suggestion is not fully visible in the screen — especially for the discontinuous scenario.

[2]The subscript in participant codes represent the stages of design exploration. i - additive change, e - single-line change, m - multi-line change (ref Table I)

Overall, this iteration of our design process acted to validate gray-text in a new context. The gray-text interface is familiar to most developers and hence, they were able to understand the suggestions even in the unfamiliar scenario of non-contiguous insertions. This points us to reuse and adapt existing interfaces in other suggestion categories (single-line and multi-line changes). Another take-away from this iteration comes from the comments made by P_i2, P_i9, and P_i10: developers will not trust or accept suggestions that they are not able to easily validate at a glance.

B. Stage II: Designs for single-line code change suggestions

In this stage of the design exploration, we investigated code-*change* suggestions, but which span only a single line. The major designs we explored in this stage are depicted in Figure 5. Here, the condition inside the if statement is originally `obstacle != null && obstacle.IsValid` and the suggestion is to change it to `ObjectNotNullAndValid(obstacle)`. The designs are described as follows:

- *Gray-text for changes.* Since we had good success with the gray-text interface for additive code changes, we explored a similar design for single-line code change in Fig 5 (c). Here, only the modified version of the code is presented to the user, with new tokens in gray text. The deleted tokens are not shown.
- *Strike-through.* We borrowed a *strike-through* based design that is already popular in AI based text suggestion tools like Grammarly [17]. In this design, existing code that will be replaced is struck out and we show a gray-text for the proposed code to the right side of the existing code (Fig 5(d)).
- *Side-by-Side Diff-view.* We also tested designs based on the traditional red-green colored diff views used in revision control software such as Git or Perforce. The side-by-side diff-view (Fig 5(a)) is similar to the strike-through view, but using colors instead.
- *In-place diff-view.* The last view (Fig 5(b)) uses red-green colors to show deleted and inserted tokens on the same line, while retaining the tokens common to both original and modified code in regular black text. For example, the token `obstacle` is common in the two version and not shown in either red or green.

For this stage, we conducted two iterations of design exploration. In the first iteration, we explored all the different types of designs mentioned above, and in the second iteration, we refined the best designs from Iteration I for special scenarios and corner cases.

1) Experiment: In the first iteration, we conducted a user study with 8 participants (P_e1 - P_e8) to test the usability of *gray-text*, *strike through*, and *diff view* designs. We integrated these designs into the IntelliCode Suggestions feature, which can detect when developers are performing repetitive code changes to multiple locations, and generate suggestions to automate the remaining ones. This feature is powered by the Blue-Pencil technique presented in [8]. Each participant had

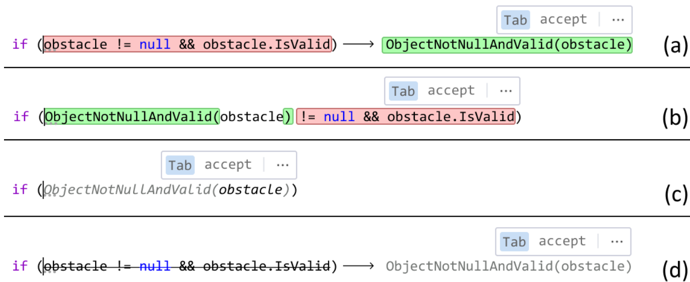

Figure 5: Designs for single-line code change suggestions: Iteration 1, (a) Side-by-side *diff-view* (b) In-place token based *diff-view* (c) *gray-text* design that directly shows the final code. (d) Strike-through design inspired by both *gray-text* and *diff-view* designs.

to perform a series of repetitive code changes in multiple locations across the file in [8]. As soon as the participant performs the second change, IntelliCode detects the repetition and when the participant goes to the next locations, IntelliCode offers a suggestion to automate the change. Participants tried all the designs using different code changes. We counterbalanced the order of the designs for each participant. Similarly, for the second iteration, we tested the usability of the refined designs for showing code change suggestions for longer lines of code by conducting a user study with 8 participants (P_e9 - P_e16). The participant performed a similar multi-location code change as the previous iteration, however, this change was longer than the previous iteration (over 80 characters).

2) Results: Of the 16 participants across both the iterations, ten participants immediately noticed and accepted the inline suggestion in the very first attempt. Five participants (P_e6, P_e7, P_e10, P_e11, P_e16) noticed the suggestion after showing it for the second time. Further, of the 16 participants, only 2 participants (P_e6, P_e10) knew that IntelliCode offered these suggestions under the lightbulbs. Still, both participants did not remember to use it. This shows that inline suggestions do not have the same discoverability issue that *light-bulb* based interfaces suffer.

With respect to the best design for showing single-line code change suggestions, all of the eight participants in the first iteration preferred the side-by-side *diff view* based design (Figure 5(a)) over designs using *gray-text* or *strike-through*. We discuss the reasons below. For longer code suggestions, where side-by-side *diff-view* cannot be rendered due to unavailability of horizontal space, five of the eight participants (P_e11, P_e12, P_e14, P_e15, P_e16) in second iteration preferred to see the suggestion rendered top to bottom, where the proposed code appears below the existing code. Next, we discuss the reasons below for these preferences.

a) Juxtaposing old and new code: In the first iteration, five of the eight participants (P_e1, P_e2, P_e3, P_e4, P_e8) explicitly mentioned they would always want to clearly see their existing code juxtaposed with the proposed code. This helps them understand what code is being replaced since they do not always remember the old code, and the juxtaposition increases participant's confidence in accepting the suggestions since they exactly understand the proposed change. Designs that directly renders the suggested code in-place as *gray-text* violates this expectation. P_e4 said "*This is very unintuitive. It doesn't even tell me what it is replacing. I'm kind of left to guess myself. I don't think I will be confident in taking this suggestion.*" Participants also want to first understand the code that will removed then look at the proposed modification. For shorter code suggestions, this can be shown side-by-side, and for longer code lines, participants preferred to see the *diff-view* top to bottom. P_e9 said "*Showing suggested code below the existing code feels more natural compared to the opposite. It's the way I read - the old code should be above.*"

b) Familiarity with diff-view: In the first iteration, five of the eight participants (P_e2, P_e3, P_e4, P_e6, P_e8) explicitly mentioned that the designs that use the diff-view based interface follows a familiar visual motif, and makes it easier to understand the red-green color coding instantly and know the code that is proposed to be removed and the code that is proposed to be added by the suggestion. P_e4 said, "*Its telling me you are going to replace the text highlighted in red with the text highlighted in green. The arrow shows that it is a replace operation.*" Three participants (P_e3, P_e4, P_e6) also explicitly referred to the familiarity with version control systems, and how colors play an important role in comprehension.

c) Mangling participants code: Participants did not like any kind of modifications (or mangling) to their current state of the code, even if transient. Six of eight participants in the first iteration did not prefer the designs that used the in-place *diff-view* (Figure 5(b)) or the *gray-text* view (Figure 5(c)), because they found that these designs mangled their code making the designs feel intrusive and confusing. P_e3 said, "*It kind of makes it confusing since you are criss-crossing the streams, you are mixing the old and new code, as compared to two different states that are clear in the previous design.*" Similarly, none of the eight participants in the first iteration preferred the design with *strike-through*, since they felt striking the code felt harsh and commanding. Participants also felt that the code editor is suggesting that the code they wrote is incorrect. P_e1 said, "*I don't want the code I have written to be struck out and be overwritten by something else, I want the suggestions to be subtle.*"

C. Stage III: Designs for multi-line code change suggestions

In this stage, we expand the designs to multi-line code change suggestions. In this stage, we conducted four iterations of design exploration. Figure 6 shows some of the designs tested in the first two iterations. As participants predominantly preferred the *diff-view* based designs in the previous stage, we explored *diff view* based designs for multi-line changes (Figure 6(a)). When a suggestion contained many lines, diff-views move the developer's existing code around a lot. To explore whether this is a real issue and how much it bothers developers, we also designed some *popup* window based designs (Figure 6(b,c)) that do not affect the code structure. Popup window based designs have been used previously to

show code from other source files or different parts of the same source file for commands such as “Peek Definition”.

During our design process, participants also expressed concerns over visual clutter when showing long suggestions. Hence to prevent user distraction we also designed a *progressive reveal* variant (Figure 6(d)), where we only show a hint bar, and the user has to initiate the hint to see the suggestion. Unlike lightbulb suggestions, these highlight the current code block where the suggestion is available inline, and is very conspicuous.

All the designs tested in the first 2 iterations are general designs for any multi-line code change suggestion. However, some participants were concerned about visual clutter for long blocks of code change shown as inline suggestions. Hence, we wanted to create a lightweight UI. But instead of focusing on any multi-line code change suggestion, we limited ourselves to a specific kind of suggestions where the code change though spread across multiple lines, is limited to only few (similar) tokens per line.

We found that for common multi-line code change patterns such as copy-paste-modify [18], where the user pastes a copied code template and edit it to suit the current context, the changes generally only include few tokens in the whole code block. Therefore, to show such suggestions inline, we took inspiration from template auto-completion supported by many IDEs including Visual Studio. Template auto-completion generates a skeleton structure of the code pre-filled with highlighted holes/voids that correspond to the core logic and the user sequentially completes these holes. For instance, Figure 7 shows a block of code defined for verifying the validity of the `obstacle` object which the user copies from another location in the code repository and pastes in the current location. They then proceed to modify the code to replace all the instances of the `obstacle` object to `player`. This code change only modifies few tokens in each line, and we can use the template interface (as shown in Figure 7(a, b)) to render the code change suggestion instead of rending the whole diff-view which is a heavier UI. To understand the feasibility of template based interfaces, we conducted two more iterations of design exploration. In the third iteration of this stage we compared *diff-view* (Figure 6(a)) and two *template* based designs. Figure 7(a) shows a template based design that indicates the actual code change using hint bar explanation, Figure 7(b) indicated the code change by automatically previewing the change directly. In the fourth iteration we refined the designs to highlight the scope of the suggestion better. Figure 7 shows the designs tested in the last two iterations of this exploration.

1) Experiment: For the first iteration, we conducted a user study with 8 participants (P_m1 - P_m8) to test the usability of the *diff-view* and *popup* designs. The participants had to perform a refactoring task for each design that involved editing multiple lines of code in multiple locations across the code file. Similarly for the second iteration, we tested the usability of the refined design with 6 participants (P_m9 - P_m14) where the participants performed the similar task as in first iteration but for a longer refactoring with over ten lines of code. For the third iteration, we conducted a user study with 11 participants (P_m15 - P_m25) to test the usability of template based designs. The participants had to perform a one copy-paste-modify change task for each design prototype. Similarly for the fourth iteration, we tested the usability of the refined designs with 8 participants (P_m26 - P_m33) where the participants performed the same task as in third iteration.

2) Results: Of the eight participants in the first iteration, five participants (P_m1, P_m3, P_m4, P_m6, P_m7) preferred the extended *diff-view* based design over the other designs, two participants (P_m2, P_m5) preferred to see the *diff-view* progressively revealed using a hint bar, and only one participant (P_m8) preferred see the suggestion in a *popup* window. We discuss the reasons below.

For very long code change suggestion, participants took long time to understand the proposed suggestion. For instance, P_m14 took around 10 seconds to read and evaluate the suggestion presented, and exclaimed “*So, first thing I'm thinking is to take it in and read the code. I'm looking at the suggestion and compare it to understand the changes.*”. Though all participants finally were able to understand and act on the suggestions, eight participants (P_m1, P_m5, P_m7, P_m8, P_m11, P_m12, P_m13, P_m14) expressed concerns that longer suggestions in other context can be distracting. P_m13 said “*I'm distrustful of things I don't understand. If I'm writing code, and boom, this thing pops up. Won't it interrupt my work?*”. Though the *progressive reveal* design could make it less cluttered/intrusive for longer suggestions, participants did not prefer to take an extra step to see the suggestion. Further, participants noted that progressive reveal option reintroduced the discoverability problem by not showing the suggestions directly in the editor. This is where our redesign using template based suggestions excelled. Seven of the eleven participants (P_m15, P_m18, P_m19, P_m20, P_m21, P_m23, P_m25) preferred the design using the default template behavior over other designs, four participants (P_m16, P_m17, P_m22, P_m24) preferred the template based design with automatic preview and no participant preferred to use the *diff-view* based design borrowed from the previous iteration. We discuss the reasons for their preference below.

a) Diff-view vs. popup view: Seven of the eight participants in the first iteration (except P_m8) did not like the suggestion shown in a popup view. Four participants (P_m1, P_m4, P_m6, P_m7) mentioned that the popup view covered the code under the window, and they wanted to see all the code lines to be confident in accepting the suggestion. For example, P_m6 said “*One issue is the popup hides the code below. If I'm making the code change, I would like to know how the code flows after the change. Showing the rest of the code is very important to understand that.*” All of the participants preferred to see red-green color coding for the *diff-view* compared to the blue based color coding. P_m4 said “*Personally I will still go with [red-green] instead of [blue] due to visibility. It is clear what is getting replaced with the way it is highlighted.*”.

b) Glanceable: In the informal interview, we asked participants what is the longest suggestion they would tolerate

Figure 6: Designs for multi-line code change suggestions: Iteration 1 (a) Extended *diff-view*. (b) *diff-view* on pop-up window. (c) *diff-view* on pop-up window with blue color scheme. (d) Progressive reveal hint bar, where the suggestion will only show after user takes an action.

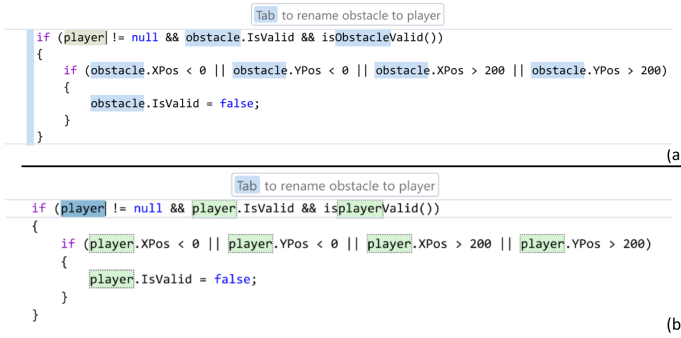

Figure 7: Designs for template based code change suggestions: Iteration 1. (a) Template based design with scope markers. (b) Template based design with automatic preview of suggestion.

within the code editor? Five of the six participants (P_m9 - P_m13) mentioned they would definitely not want to scroll to see all of the suggestion, and any suggestion shown on the screen should be glanceable. P_m11 said "*As I said, it's a matter of seeing it in one glance. If it's too long and I have to scroll to compare, then it will take more time to review.*" Participants also noted that for very long suggestion, they also have to keep lot of code in memory to compare — which affects suggestion comprehension.

c) Diff view vs. template based design: None of the eleven participants preferred to use *diff-view* compare to the template-based prototype due to three main reasons. 1) Six Participants (P_m15, P_m16, P_m18, P_m21, P_m22, P_m24) felt the UI was busy, cluttered, and overwhelming. 2) Three participants (P_m17, P_m20, P_m23) mentioned that they did not obtain any new information from [diff-view] to justify the added complexity. P_m20 said "*I don't feel like it gave me any extra information with all that extra code. With [the template design], everything was highlighted, and I know what was going to happen. That was enough information.*" 3) Five participants (P_m18, P_m19, P_m20, P_m23, P_m25) felt like the suggestion was duplicating code when seeing the *diff-view*.

d) Reference to template experience: Nine participants immediately compared their experience of doing the task to the template interface experience present in Visual Studio when they saw either of the template based designs. The familiarity with the template experience made it easy for the participants to understand the proposed changes. P_m21 said "*My first though is that it mimics the functionality of templates. It triggered familiarity with templates. I use it a lot. I understand what it says.*" Similarly P_m18 said "*I'm seeing the highlight I usually see for rename refactoring [which uses the template interface]. I think I can just start typing.*". However, four participants (P_m16, P_m18, P_m19, P_m23) expressed some confusion over the scope of the suggestion. For instance, P_m23 incorrectly claimed "*I'm only expecting this to happen over the function / method I'm in.*" though the scope was just in the pasted code. We addressed this in the designs explored in the last iteration by adding a horizontal scope marker (Figure 7(a)).

e) Automatic preview: Seven of the participants who preferred the default template design over the design with automatic preview mentioned they do not want to see other locations automatically changed. Since automatic preview renders over the old code, they cannot determine if the proposed change at the given location is valid. P_m18 said "*I would rather prefer [diff-view] over [automatic preview], because the [automatic preview] was not giving me the option to see what was there before.*"

IV. Field Deployment

We selected the most promising interfaces tested in our design exploration to be fully implemented and tested in

production. As of this writing, we were able to successfully deploy the interface we chose in Stage 1 in Visual Studio 2022. Additionally, we have released the interface selected in Stage 2 in Visual Studio 2022 Preview, which is the preliminary version of Visual Studio's next release. We share the field deployment usage numbers in this section.

1) Inline code insertion suggestions: We have released the *gray-text* interface for all code changes produced by IntelliCode that can be classified as code insertions in Visual Studio[3]. This includes not only the code changes we tested in our user studies (Figure 4) but any other repetitive code change suggested by IntelliCode that just inserts code. The feature has been used by hundreds of thousands of developers every month. The release had a big impact on our usage numbers. Even though code insert suggestions correspond to only 25% of the code change suggestions produced by IntelliCode, its usage with the gray text interface corresponds to 60% of total usage of code change suggestions. Additionally, this interface led to a 3.5x increase in regular users of the feature.

2) Inline single-line code change suggestions: We deployed the *diff-view* experience (Figure 5(a)) for the single-line code change suggestions in Visual Studio 2022 Preview. We conducted an A/B test over the course of 2 weeks to evaluate the impact of this new feature. Over this period, several thousands of developers used the feature, which lead to an 176% increase in code change suggestions accepted by users and 29% more regular users.

Both deployments show a significant positive impact on the usage of IntelliCode because of the new user experience based on inline interfaces.

V. Design Principles for inline code suggestions

Informed by the qualitative data from our design exploration, we inductively propose five design principles for future tools that aim to implement inline code suggestions (Table II). For some of the design principles, we also draw parallel to design principles from *cognitive dimensions of notations* [19], which are highlighted using `this typeset`.

A. Glanceable suggestions

With code editors continuously adding many code suggestion tools, and especially AI powered tools that can dynamically suggest for endless scenarios, it is not feasible for the user to keep track of all the available code suggestion tools. Tying to guess all the refactoring a user can expect in the lightbulb in the current context adds a lot of cognitive load. For the suggestion to be discoverable, it needs to be glanceable. This principle is consistent with cognitive dimensions of notations by increasing `visibility` and decreasing `viscosity`. Inline code suggestions are glanceable by being directly rendered in the code editor, taking the user's guess out of the equation. Of the 61 participants in the user study, only two participant knew that the suggestion was already present in the lightbulb which they could have used to perform the task. Whereas, 53 of the 61 participant immediately noticed and accepted the inline suggestion.

> P_e12: I rarely use lightbulb since most of the time, nothing exists for the things I want to do. With what you are showing I have to go through two menus to see that there are 5 possible edits. I will never know to look and I would normally just do it myself

> P_e11: ... [with inline suggestions] Its very quick just press this button, versus me having to think about oh I want to do this via a tool, and I'll have to click a lightbulb and scroll through to identify which one is going to do this for me.

[3]https://devblogs.microsoft.com/visualstudio/just-in-time-refactoring-intellicode-suggestions/

Design Principle	Explanation
Glanceable Suggestions	Suggestions should be proactively visible to the user to make it easy to discover and take action.
Juxtaposition	The suggestion should explicitly indicate and juxtapose the existing code affected by the suggestion and the proposed code to improve user comprehension and support quick action.
Simplicity through familiarity	Reusing existing familiar interface elements to indicate proposed changes reduces visual clutter, improves comprehension, and reduces cognitive load.
Sufficient Visibility for validation	The users should be able to see the whole suggestion when taking the action to prevent premature-commitment.
Snoozability of suggestions	The user should be able to snooze inline suggestions to prevent interruptions when user intends.

Table II: Our design principles for inline code suggestions.

B. Juxtaposition of original and suggested code

For the users to understand the proposed suggestion, they need to clearly understand 1) the scope of the code being affected by the suggestion, 2) what the affected code is, and 3) the newly proposed code. It is very important to *juxtapose* the original and the suggested code to help the user compare and evaluate the suggestion. Without understanding the original code that is being affected, the user is forced to `prematurely commit` without fully understanding how it affects the code. Prior research has shown programmers want to juxtapose code during various tasks [20], [21]. This juxtaposition also increases trust in the tool by making all the changes explicitly visible, helping the users take appropriate decisions. In designs where we failed to highlight or overrode the original code with the proposed edit, participants were more confused and took more time to understand the change due to the added `hidden-dependency`. To work around this, participants generally accept the code and perform an undo to see the original code, and repeat this cycle a few times to understand the suggestion.

> P_e2: By seeing the old code, I can easily understand what is being changed, otherwise I'll have to press undo and redo to compare with the previous code. I can't trust the suggestion without seeing what is going to change.

> P_e1: By automatically replacing my code it feels like the tool is forcing it upon me. Don't tell me what to do, support me with what I do.

We achieve this by using diff-view based designs where the original code and the suggested code are juxtaposed either side-by-side or top-to-bottom, helping the user to compare and evaluate the suggestion. We also make use of the familiar red-green color coding to help with the comprehension, following the `role-expressiveness` principle.

> P_e6: When you are checking code, and you are doing a file compare, things are in red and green, and you know exactly what has changed. .... The colors are really standard, and it really easy to see that. The other designs aren't quick.

C. Simplicity through familiarity

Code editors are visually busy interfaces. At any given time there are many window panes showing errors, warnings, debugging information etc. The actual code also has visual artifacts like syntax highlighting, reference information runtime information etc. It is vital to reduce the visual clutter and visual redundancy added by our inline suggestion. This can be achieved by reusing familiar visual colors and interfaces that elicit the same understanding as their original use. When designing for the diff-view, we only highlight part of the expression that is actually changing and help the user to understand the change implicitly by using the familiarity of the red-green color coding used in source-control interfaces, thereby reducing the visual clutter (akin to `role-expressiveness`).

> P_e13: We have a long line of code and it is telling me just the code that is being changed. I don't have to focus on the whole line of code, and just focus on the part that is changing.

> P_e15: I don't like highlighting the whole line. It makes sense to just show what the change is

What if we cannot afford to juxtapose all the new code due to space constraints or visual clutter? When applying the *Juxtaposition* principle for multi-line code change suggestions, instead of rendering the new code completely, we exploited the users implicit understanding of the familiar template interface and just showed a hint bar explaining the change. This allowed the user to compare and evaluate the proposed suggestion without having to materially see the whole proposed code.

> P_m19: The highlight means that, the tool went through the rest of the code in scope, and is suggesting everything else that is to replaced.

> P_m21: My first thought is that it mimics the functionality of templates. It triggered familiarity with templates. I use templates a lot.

Instead, in the designs where we did not exploit this familiarity and rendered the all the code via diff view, participants felt they did not gain any new information with the added visual clutter, and it just made them feel overwhelmed.

> P_m20: ...for this scenario why would you show [*diff-view*] when the [template-based] designs give all the information in a simpler view.

> P_m21: To me this is a little too busy, because it is taking my eye away from where the code is, and I'm losing the context in a way.

D. Sufficient visibility for validation

For the users to make an informed decision and prevent `premature commitment`, all the information should be visible to the user. Though in retrospect this sounds obvious, there are some subtle visual expectations from the users.

> P_m9: I like it to be verbose than guesswork, so the more information you show its better. I clearly don't want to refactor something without understating all the changes.

Participants did not like the designs where they have to horizontally or vertically scroll to see the entire suggestion. Moreover, many participants also mentioned they would simply ignore the suggestions if it takes them more than a few seconds to understand and act on it.

> P_m11: As I said, it's a matter of seeing it in one glance. If it's too long and I have to scroll to compare, then it will take more time to review.

> P_m12: I'm okay for multiple lines, but it will take me a lot more time to review longer code change. But if we really have to scroll, then it will not be fine for me. I might lose understanding of what I was working on. That is not okay.

Participants also want to see the context surrounding where the suggestion will be inserted.

> P_m6 said One issue is the popup hides the code below. If I'm making the code change, I would like to know how the code flows after the change. Showing the rest of the code is very important to understand that.

E. Snoozability of suggestions

Though inline suggestion improve the discoverability of code edit tools, it can be interrupting if the user is in code authoring mode and has no intent to edit or refactor the code. It is vital to understand the users intent. Though AI models are now starting to understand user's editing patterns, it is still difficult to understand user's intent. In these scenarios, seven participants requested a snooze feature where they can pause the inline suggestions for a period of time or for the current session. Some participants also suggested to vary the frequency of inline suggestions based on their recent acceptance rate. For example, P_e14 asked for a way to suppress suggestions and P_e10 asked for a 30-minute snooze feature.

They also wanted to be able to pick and choose the specific code edit tools supported by the inline suggestions. Though prior work has shown majority of the users do not wish to configure refactoring tools [10], it can be useful for power users who wish to tune it to their preference.

P_i1: Since I'm seeing it for the first time, I'm fascinated - if it pops up frequently, or things I don't want to do then I might turn it off. If there is an option to control where all I can see, then I will be using it appropriately.

Making the suggestions easy to dismiss or ignore is another way to reduce the distraction. Participants mentioned if we make the suggestions less prominent, they can ignore when they do not want to.

P_e7: I think the experience is positive as long as the suggestions is just out of the way, ideally a little muted. Many times it may think it is smart, but it's not. So make it less prominent.

VI. Related Work

Modern code editors support many tens of code suggestion tools, and are constantly updated with more tools with every passing year. Code suggestion tools provide numerous benefits in improving the speed and accuracy of creating and maintaining software [22] by helping programmers prevent code smells, make the code modular, migrate to new API versions and more [23]. In fact Fowler [23] catalogs 72 code suggestion (refactoring) tools that can produce significant benefits to code that are now available as a part of any modern IDE. It is a widely known fact that professional programmers refactor their code frequently [22]. However, these tools have been found to be severely underused by programmers [10], [11], [24]. By analyzing 240,000 tool usages in Eclipse, Murphy-Hill et al. [10] found that only 8 of 23 code suggestion tools surveyed have any significant usage, with *RENAME* refactoring accounting for over 28% of usage. One of the main reasons for this under-use is due to the poor usability of these tools [10], [11]; *RENAME* tool may be performed most often due to the simplicity of the user interface. We discuss the different aspects of usability that affect the usage of code suggestion tools in this section.

Discoverability: Currently these code suggestions shown via menus like the *lightbulb* icon (Figure 2) in Visual studio, analogous to Quick Assist in Eclipse, are not discoverable [25]. The user can only know about suggestions offered in the quick-assist menu by frequently checking the menu or learning about its features from other sources. In general, the lack of awareness of all the code change tools present in the code editor leads to disuse of the tool [10], [25]. By *silently* overloading the existing suggestions menu with more and more code edit tools exacerbates the discoverability problem, since people don't know or expect to see these suggestions in the current interface. In our work, we show that in-line suggestions can overcome this discoverability problem by proactively showing code suggestions in-line in the code editor.

Context-switching: Prior work has also found that affordances of current tools force the user's attention away from the task [10]–[12]. Murphy-Hill et al. [11] found that the context switching is required to manually initiate the code edit tools, and this process distracts the programmer from their primary programming task. This has a particularly strong effect when the suggestions are hidden in nested menus. Using a hot-key might seem to be an ideal way to speed up the initiation, but hot keys can be difficult to remember [26]. In our work, the use of in-line code suggestions precludes the context switching by directly showing the suggestions in the editor.

Cognitive preconditions: There are several cognitive preconditions (**CP**) required for a successfully applying a code suggestion [12] First, the programmer must realize they need to perform an edit [**CP1**]. Second, they should know the support for such an edit exists [**CP2**]. Third, they should know that the code edit is applicable in the current context [**CP3**]. Fourth, they should believe that initiating and accepting the code suggestion is faster than performing it manually [**CP4**]. Fifth, they must trust that the suggestion will behave as intended [**CP5**], and finally, they must be willing to context-switch away from writing code to invoke the tool [**CP6**]. In our work, we found that showing code suggestions in-line helps the user by supporting the first three cognitive preconditions ([**CP1-CP3**]) by proactively showing the suggestion *in the editor*. In-line suggestions also make it easier for users to evaluate and [**CP5**] by juxtaposing the current code with the proposed suggestion. Finally in-line suggestions makes [**CP6, CP4**] irrelevant since the action of accepting or rejecting the suggestion is instant.

In addition, with the advent of highly sophisticated code suggestion tools powered by AI, code editors can now offer smarter refactorings by learning programmer's code edit patterns on the fly [16], [27]. These tools have the potential to significantly improve developer productivity. However, currently, these AI assisted code suggestion tools continue to piggyback on existing editor interfaces which still suffer from the usability problems mentioned above. It is vital to improve the usability of refactoring interfaces.

VII. Conclusion

In this paper, we presented a systematic design exploration for effectively showing AI-based code suggestion using inline interfaces. Through exploratory user studies and usage data from field deployments, we show that inline suggestions overcome the discoverability problem faced by existing code suggestion tools by meeting the users in their flow. The field deployments show a drastic increase in user reach and acceptance rate demonstrating the potential of inline code refactorings. Informed by the qualitative data form our user studies, We also propose five key design guidelines for future tools that aim to implement inline suggestions for any code editor. Moving forward, we hope that the design principles we proposed will help future code suggestion tool builders as their foundation for designing inline suggestions. With the recent advancements in AI powered code suggestion tools, we believe our designs and our design principles will be timely and help fill an important gap that enables tool builders to explore impactful ways of empowering programmers around the world.

Acknowledgment

We sincerely thank all the members of Microsoft PROSE, Microsoft Roslyn, and Microsoft Intellicode teams for their support and resources provided to make this research possible. This work was partially funded by NSF grant 2107391. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] "Visual studio 2022," https://visualstudio.microsoft.com/vs/, 2022, accessed: 2022-9-13.

[2] "Visual studio intellicode: Visual studio," May 2022. [Online]. Available: https://visualstudio.microsoft.com/services/intellicode/

[3] "Github copilot - your ai pair programmer," Jun 2021, accessed: 2022-1-8. [Online]. Available: https://github.com/features/copilot

[4] "Top 10 features of visual studio 2022," May 2022. [Online]. Available: https://inceptivetechnologies.com/blog/top-10-features-of-visual-studio-2022/

[5] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, 2022, pp. 1–7.

[6] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *arXiv preprint arXiv:2206.15000*, 2022.

[7] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Can openai codex and other large language models help us fix security bugs?" *arXiv preprint arXiv:2112.02125*, 2021.

[8] A. Miltner, S. Gulwani, V. Le, A. Leung, A. Radhakrishna, G. Soares, A. Tiwari, and A. Udupa, "On the fly synthesis of edit suggestions," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[9] X. Gao, S. Barke, A. Radhakrishna, G. Soares, S. Gulwani, A. Leung, N. Nagappan, and A. Tiwari, "Feedback-driven semi-supervised synthesis of program transformations," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–30, Nov. 2020.

[10] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 5–18, Jan. 2012.

[11] E. Murphy-Hill and A. P. Black, "Making refactoring tools part of the programming workflow," *http://citeseerx.ist.psu.edu > viewdoc > summaryhttp://citeseerx.ist.psu.edu > viewdoc > summary*, 2008.

[12] S. R. Foster, W. G. Griswold, and S. Lerner, "WitchDoctor: IDE support for real-time auto-completion of refactorings," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, Jun. 2012.

[13] J. Nielsen, "Parallel & iterative design + competitive testing = high usability," https://www.nngroup.com/articles/parallel-and-iterative-design/, Jan. 2011, accessed: 2022-9-10.

[14] L. Molin, "Wizard-of-Oz prototyping for co-operative interaction design of graphical user interfaces," in *Proceedings of the third Nordic conference on Human-computer interaction*, ser. NordiCHI '04. New York, NY, USA: Association for Computing Machinery, Oct. 2004, pp. 425–428.

[15] P. Groenewegen, "Discover quick actions for common tasks as you type, with intellicode," https://devblogs.microsoft.com/visualstudio/discover-quick-action-intellicode/, accessed: 2022-10-13.

[16] Y. Zhang, Y. Bajpai, P. Gupta, A. Ketkar, M. Allamanis, T. Barik, S. Gulwani, A. Radhakrishna, M. Raza, G. Soares, and A. Tiwari, "Overwatch: Learning patterns in code edit sequences," *Proceedings of the ACM on Programming Languages*, no. OOPSLA, Jul. 2022.

[17] "Under the hood of the grammarly editor, part two: How suggestions work," https://www.grammarly.com/blog/engineering/how-suggestions-work-grammarly-editor/, Feb. 2022, accessed: 2022-9-11.

[18] K. Narasimhan and C. Reichenbach, "Copy and paste redeemed (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 630–640.

[19] T. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1045926X96900099

[20] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, "Code bubbles: Rethinking the user interface paradigm of integrated development environments," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 455–464. [Online]. Available: https://doi.org/10.1145/1806799.1806866

[21] A. Z. Henley, S. D. Fleming, and M. V. Luong, "Toward principles for the design of navigation affordances in code editors: An empirical investigation," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ser. CHI '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 5690–5702. [Online]. Available: https://doi.org/10.1145/3025453.3025645

[22] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE Softw.*, vol. 25, no. 5, pp. 38–44, Sep. 2008.

[23] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[24] A. P. Black, "Better refactoring tools for a better refactoring strategy," *IEEE Software*, 2008.

[25] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, Jun. 2012, pp. 233–243.

[26] T. Grossman, P. Dragicevic, and R. Balakrishnan, "Strategies for accelerating on-line learning of hotkeys," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '07. New York, NY, USA: Association for Computing Machinery, Apr. 2007, pp. 1591–1600.

[27] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "CODIT: Code editing with Tree-Based neural models," *IEEE Trans. Software Eng.*, vol. 48, no. 4, pp. 1385–1399, Apr. 2022.