A Tour Through Code:
Helping Developers Become Familiar with Unfamiliar Code

Grace Taylor
Microsoft
grace.taylor@microsoft.com

Steven Clarke
Microsoft
steven.clarke@microsoft.com

Abstract
Becoming familiar with an existing code base can be a challenging and time-consuming process. A developer who is new to a code base and needs to implement a new feature or fix a bug might browse and search the code base for locations providing good starting points for their task. From these starting points they may then navigate to other locations in the code base by traversing semantic and other relationships in the source code that may be relevant to their task. Although most developer tools provide ways to discover some of these different relationships, the relevance of specific relationships can often only be determined after traversing the relationship. This often necessitates some amount of backtracking as a developer navigates one relationship to a new destination in the source code, only to find that the destination is not relevant to their task. Furthermore, many such relationships between different locations in source code cannot easily be determined by developer tools alone. This paper presents the results of a study that investigated the use of CodeTour, a tool that enables the creation and presentation of annotated code tours through a code base. Such tours can indicate relationships between locations in a code base that are relevant to a task, but which could be difficult to identify using regular features found in developer tools. The paper describes the extent to which these code tours helped developers find the source of bugs and compares the experience with those who attempted the same task without the tool.

Introduction
Every time a software developer joins a new project or forks a new repository, they need to become familiar with the code in that project or repository. This onboarding process can involve a considerable amount of time, money, and effort (Dagenais et al., 2010, Begel & Simon, 2008). The result is that it can take some time before a new developer feels productive. One way to help developers onboard to a new codebase is to provide up-to-date documentation. However, even though up to 11% of the cost of a software development project can be attributed to documentation (An Overview of the Software Engineering Laboratory, 1994), the resulting documentation might not provide what developers really need (Aghajani et al., 2019).

Documentation content is often incomplete, outdated, or incorrect (Aghajani et al., 2019). In some cases, documentation may even lack any useful content at all. For example, an examination of documentation across two large frameworks indicated that a high proportion of the documentation contained no information and merely rehashed API names (Maalej & Robillard, 2013).

Resource constraints significantly impact a developer's ability to write documentation. (Stetting and Hiejstek, 2011) reported that developers they surveyed were able to spend on average 15 minutes per day creating documentation for their projects. Many implied that they felt this was insufficient time to spend on documentation, saying that they felt there was too little documentation available for the projects they work on.

The type of documentation that developers create and the type of documentation that they need can vary. Code comments are one of the most common types of documentation and are lightweight in nature, but often lack substance (Aghajani et al., 2020). On the other hand, tacit knowledge is considered crucial yet consistently missing from documentation. For example, knowledge related to the performance of the software or the environment in which it runs, differences between different versions etc is not often found in documentation, yet developers find this kind of information important for their work (Maalej & Robillard, 2013, Robillard & DeLine, 2010).

Documentation does not always have to come in the form of written text. In some cases, mentorship can be effective (de Janasz et al., 2003), but searching for the right mentor costs time. But even once the right mentor is found, they may not always be available to share their insights and expertise (Ko et al., 2007).

Understanding the ways that developers seek information about a new codebase can help identify reasons why existing documentation might fail. For example, (Sillito et al., 2008) lists the questions that developers ask themselves when learning a new codebase. In many cases, the answer to the question is found by piecing together multiple pieces of information that are found in multiple locations throughout the codebase (Ko et al., 2007). One code comment alone often does not suffice.

But stitching together multiple sources of information to answer one question can be difficult. Navigating between the disjointed external documentation and multiple files inside the codebase adds unnecessary complexity (Robillard & DeLine, 2010). Furthermore, there may be differences in the ways that different developers approach an information seeking task. Some strategies may be more likely to lead to success than others.

When working on certain programming problems, developers may be faster at finding information they need if they are more conscious of the different problem-solving strategies that they adopt (Loksa et al., 2016). Tools that prompt developers to ask and answer a different range of questions when learning code may help those developers learn about and onboard to the codebase independently.

Our insights about how developers onboard to a new codebase have also been informed through informal conversations we have had with developers. In those conversations we learned about what those developers have done to help others onboard to a new codebase. We spoke with eight developers who had used CodeTour, an extension for Visual Studio Code, which allows developers to create one or more 'tours' within a codebase, where a tour is a linked list of nodes, with each node containing a location in the source code (file name and line number) and text. We wanted to learn more from these developers about the kind of tours they created and the impact they had on the onboarding process.

The developers we spoke to suggested that code tours provided a better way to communicate tacit information such as little-known quirks around compiling and building the codebase. They also suggested that code tours help identify important locations in the codebase and the relationships between those locations, helping them become more familiar with the codebase.

This has motivated us to explore if code tours can address the learning and onboarding challenges identified by Robillard & Deline (2010), Ko et. al (2007) and Sillito et. al (2008) when developers collect information from multiple sources through the codebase.

In this paper we describe CodeTours, an extension for Visual Studio Code that allows developers to create tours in a code base. Then we describe the results of a usability study that aimed to validate the following hypotheses:

H1: Code tours provide a useful starting point for people orienting themselves to a new code base

H2: Code tours encourage developers orienting to a new code base to explore more of the code base than they would have done otherwise

CodeTour

CodeTour is a documentation tool that allows developers to create code tours within Visual Studio Code, a popular code editor. A code tour is a linked list of two or more nodes, where each node contains a location (file name and line number) in the source code and some descriptive text. CodeTour visualizes nodes in a tour by drawing an icon in the left margin of the editor adjacent to the specific location.

When that node is selected, the associated text for that location opens in a separate window within the code (see Figure 1).

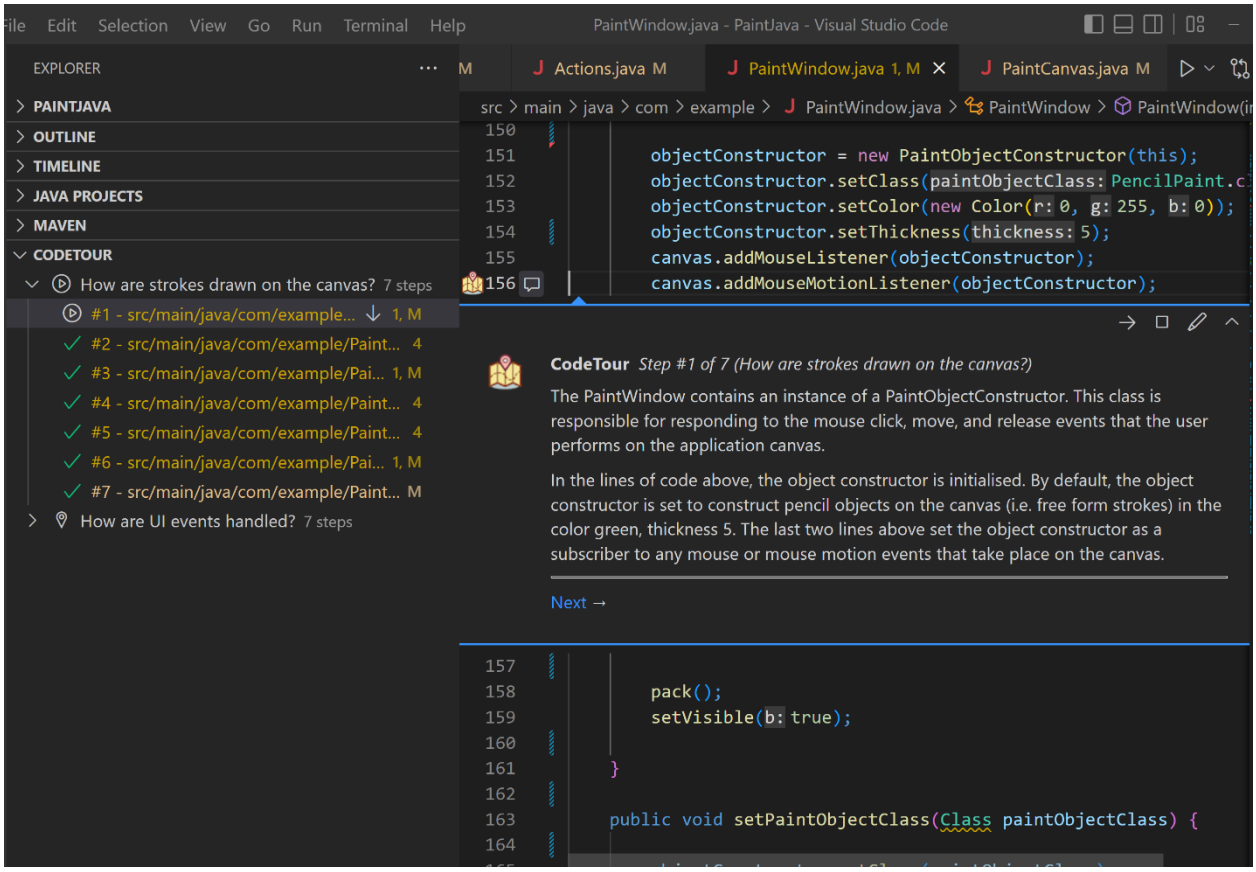

Figure 1 A screenshot from VS Code showing a step within a CodeTour opened in the editor on the right hand side and the full table of contents for the tour on the left hand side.

A table of contents view allows developers to view every node in the tour and to navigate to any of those nodes (see Figure 1).

Related Work

Multiple tools and prototypes have been developed that tackle different aspects of orienting to a codebase. Chat.code (Oney et al., 2018), for example, is a project that allows developers to text chat with each other inside the codebase, where messages are persisted to lines of code. Developers can reference current or previous versions of code in their chat messages and respond to each other over time. Documentation does not need to be kept up to date because developers can follow chat threads to learn about changes. Chat.code allows developers to reference previous versions of code in chat threads to tell a story about the evolution of the code. This helps with rich contextual information about code but does not provide threads throughout code.

Codepourri (Gordon & Guo, 2015) is a project that crowdsources annotations inside code by prompting users to compare and write annotations, and visualizing its execution state. The system asks all learners who are reading the tutorial to contribute annotations and vote on the best annotation to display. Instructors can create and share tutorial links with students. A given tutorial will remain up to date when users are incentivized to share and vote on annotations. The study authors write that users are more likely to contribute annotations when there are already existing annotations in the tutorial. Since the content is crowdsourced, the tutorial increases in quality when a higher volume of learners contribute to voting and writing. However, this tool does not address quality or trustworthiness issues for new tutorials who do not have many users.

Other projects, such as Adamite (Horvath et al., 2022), are developing tools that allow developers to leave annotations throughout a document without having to write code comments. Adamite allows

developers to categorize their own annotations, and when researchers coded the annotations from authors, they found that many of the annotations were redundant. The researchers also found that a lot of developers used the tool as a mechanism to leave notes to themselves. Adamite shows points of interest inside documentation dynamically, but relies on crowdsourced information to display accurate information. We found that some developers used CodeTour to leave notes to self, but this will not be the focus of this paper.

In the area of learning new technology from a vendor or external company, Codelets (Oney & Brandt, 2012) allows developers to copy a block of sample code containing interactive code context into an editor. It also proposes a "builder interface" and a "helper" in code which could write most of the code for the user, but requires them to set parameters. After the user copies the sample code into the editor, they can view a description of the snippet along with helper components, which aim to help the user understand and integrate the sample code into their own work. As a result, learners in the treatment group spent more time customizing their pasted sample code before running it, compared to the control group. Some developers copied and pasted sample code before understanding its explanation, which has the risk of introducing errors in one's code. Codelets is most helpful as part of sample code documentation on a website where users assume the resource is trustworthy. However, Codelets does not support editing or sharing annotations with the sample code.

Some tools focus on providing different views of code. Code Maps (DeLine et al., 2010) creates diagrams that represent a codebase for different scenarios in hopes that new developers may better understand the codebase. The researchers generated these Code Maps by collecting examples of developers' visual representations of different codebases, including sketches and diagrams, to find conventions for representing a codebase. It shows relationships between artifacts and pieces of a codebase, but it does not show paths or points of interest inside code. While CodeTour shows different views of the codebase through paths inside code and a table of contents, its focus is to share tacit knowledge through text, not create diagrams to describe relationships.

Other tools focus on showing threads between developer artifacts. For example, Codebook (Begel et al., 2010) describes a broad relationship analysis framework, which shows links between different artifacts such as work items, emails, and code files. The researchers implemented Codebook in two contexts, in an artifacts search, where users can search for artifacts along with the people who can be helpful, and in Deep IntelliSense, where users can trace that block of code's change history and the people associated with that artifact. Codebook attempts to solve software team coordination problems by allowing developers to trace threads of interaction between team mates, but does not show paths throughout a codebase.

Another tool, Team Tracks (DeLine et al., 2005), is a recommender system that automatically identifies relationships throughout code by recording how frequently users navigate a certain path throughout a codebase, so that the system can guide new developers by recommending popular code paths. The Team Tracks recommendation system automatically generates a path without a description or context, whereas CodeTour's paths are manually defined but offer detailed descriptions.

Method

In this study, our aim was to determine if CodeTour provided value to developers when working with an unfamiliar code base. We did not study how easily CodeTours could be created, or indeed how motivated developers would be to create CodeTours in the first place. We believe that without evidence that CodeTours would provide any value on top of existing annotation and code commenting practices, it would be difficult to motivate developers to create CodeTours.

To investigate the effect of using CodeTour to become familiar with an unfamiliar code base, we created two code tours for a Java code base that implements a simple Paint program (the same code base that was used in Amy Ko's work on the Whyline debugger – Ko and Myers, 2008). The code base consists of 8 source files and a total of 598 lines of code (including white space). The application is implemented using the standard AWT and Swing libraries. The tours provide annotated paths through distinct locations in the code base. Each tour describes how the actions that a user takes results in strokes being

drawn on the main canvas and how UI events result in different commands being invoked in the application.

We recruited fifteen participants to take part in the study. Each participant had a minimum of two years professional experience with Java and spent at least 20 hours per week writing Java code. Each participant was given access to a computer that was set up with a code editor, the code base, and the Java runtime. They were asked to work on two tasks, thinking aloud, while we observed remotely:

- Task 1: The undo button does not undo the last stroke drawn on the canvas immediately. Find out the cause of this bug.
- Task 2: The tool to draw straight lines on the canvas has not been implemented. Write code that implements this tool.

One group of seven participants was provided with the two code tours, while the other group of eight participants was not. The code tours did not explicitly describe answers to either of the tasks but provided information about how UI events are handled in the application and how strokes are drawn on the canvas.

Participants were given one hour to work on these tasks. Participants worked on the tasks in sequence, not starting the second task until they completed the first task or until they had spent approximately 30 minutes on the first task. They were permitted to use any of the tools available to them in the code editor such as search, goto definition, find all references, and the debugger. As they worked, they were asked to talk aloud to describe what they were thinking as they worked.

Participants who were given access to CodeTour were presented with a two-minute demo of the functionality before they started work on the task. These participants were told to make use of the code tours while working on the tasks.

Measures

The audio and video (showing the code editing screen that participants worked on) of each session was recorded. Before performing each task, we asked participants how confident they were that they would be able to complete the task successfully. Participants responded using a five-point Likert scale where the lower value (1) represented not confident at all and the higher point (5) represented completely confident. At the end of each task, we also asked them how well they thought they understood the codebase. We used a similar scale with 1 representing no understanding and 5 representing complete understanding.

We recorded the time that each participant spent on each task, starting the timer when participants first started reading the task instructions and stopping when they told us they were either satisfied that they had completed the task or had reached a point where they were unable to continue. Due to differences in how participants carried out each task, in particular the extent to which they talked out loud while working, the time spent working on tasks does not serve as a uniquely reliable indicator of the impact of CodeTour, but combined with other measures it can provide some useful insight.

The files that participants opened while working on the tasks, the number of times these files were active in the editor, and the amount of time that each file was active in the editor were recorded.

Results

In this section, we will refer to participants in the CodeTour group as (ct*n*) and participants in the non-CodeTour group as (noct*n*).

Due to the small sample size, none of our results are statistically significant. However, we do believe that our observations raise some interesting questions worthy of discussion and invite further exploration.

Table 1 presents the success, confidence and understanding scores for all participants across both tasks. Seven participants were successful in task 1, and two participants were successful in task 2.

	Task 1			Task 2		
Participant	**Success**	**Confidence**	**Codebase understanding**	**Success**	**Confidence**	**Codebase understanding**
ct1	Yes	5	4	No	4	3
ct2	No	4	4	No	*No report*	3
ct3	No	3	4	No	3	3
ct4	Yes	5	*No report*	No	5	4
ct5	Yes	3	3	Yes	3	4
ct6	No	2	3	No	*No report*	3
ct7	Yes	5	4	No	3	5
noct1	No	2	2	No	1	2
noct2	No	2	3	No	2	2
noct3	No	2	*No report*	n/a	*n/a*	*n/a*
noct4	No	5	*No report*	No	3	*No report*
noct5	Yes	4	3	No	*No report*	3
noct6	Yes	4	3	No	3	4
noct7	No	3	2	No	2	3
noct8	Yes	3	*No report*	Yes	2	*No report*

Table 1 Success, confidence and understanding scores for both groups of participants across both tasks.

We did not see any major difference in success across both tasks. The ct participants were slightly more successful in task 1 but had the same success rate in task 2. However, we did see a difference in terms of participants' self-assessment of how well they understood the codebase after each task.

Given the small sample size it is not possible to point to any meaningful difference in the confidence or understanding scores between the ct and noct participants. From our own observations though, participants in the ct group performed slightly better when identifying areas of concern in code for explaining the undo bug.

Identifying cause of bug in task 1 – ct group

For the first task, the crucial code is the undo() method exposed by the PaintCanvas class, contained in the file PaintCanvas.java. This code is called by the PaintWindow.undo method which is in turn called by the undoAction.actionPerformed method defined in the Actions class. Figure 2 shows the sequence of calls that result in PaintCanvas.undo() being called.

Figure 2 The sequence of method calls resulting in PaintCanvas.undo() being called.

During the first task, we observed the CodeTour participants using the code tour to follow paths around and within the undo() method. The code tours do not describe how the undo functionality works but they do describe how other similar functionality works.

As a result, we observed CodeTour participants focusing their efforts on the files that were contained in this path whereas the non CodeTour participants spent time looking in other files that were not related to the task. Table 2 shows the average number of times that a file was active in the editor (active means that focus was placed inside that editor) across all participants. For this task, the files Actions.java, PaintWindow.java and PaintCanvas.java were crucial to being successful since these contained the implementations of the actionsPerformed() and undo() methods, highlighted in Figure 2. The table shows that the files Actions.java and PaintCanvas.java were most active for those that used CodeTour and for those that did not. Both groups spent a fair amount of time in PaintCanvas.java. However, the noct group activated this file 10 times whereas the ct group activated this file 5 times, which implies that the noct group spent more time opening this file then focusing attention on another file, then returning.

In addition, the table shows that files such as EraserPaint.java and PencilPaint.java, which are not relevant to the task, were not active at all for CodeTour participants. Those that did not use CodeTour though spent some time browsing the code contained in these files.

Task 1: Average time and number of times active in each file per participant				
	Did not use code tour		**Used code tour**	
Time on task	26:31		23:35	
	Time in File	**Active**	**Time in File**	**Active**
Actions.java	04:23	6	03:40	6
EraserPaint.java	00:20	2	00:00	0
PaintCanvas.java	09:34	10	08:14	5
PaintObject.java	00:13	2	00:02	1
PaintObjectConstructor.java	01:31	2	02:10	2
PaintObjectConstructorListener.java	00:01	1	00:00	0
PaintWindow.java	07:05	10	06:13	10
PencilPaint.java	00:18	1	00:00	0

Table 2 Number of times that a file was active in the editor while participants worked on task 1.

One CodeTour participant, ct1, was able to correctly explain why the bug occurs and describe the logic for addressing the bug.

Figure 3 shows the code that contains the bug. When referring to this code, participant ct1 correctly identified the problem, saying "*when it removes the last element from paint object, it should try to repaint it again*".

```
public void undo() {

    paintObjects = history.lastElement();
    history.removeElement(history.lastElement());
```

Figure 3 The code for the PaintCanvas.undo method.

The other CodeTour participants, ct2 and ct3, followed similar paths to and around the undo() method but never fully realized the cause of the issue. They spent time focusing on the history, lastElement, and

removeElement methods. They believed that the bug was within code that was already present in the function and did not consider that the bug might have occurred due to missing code. The code tour that they had followed had described how similar functionality works in the application and had highlighted calls to repaint when they occurred. However, even though they had browsed this tour and read the methods implementing similar functionality, they were not aware of the significance of the repaint() method.

Utilizing the CodeTour

We observed some differences in the way that participants used the CodeTour.

Participant ct1 was the only participant who was able to complete task 1 successfully. They spent the first 4.5 minutes of the task reading through the first CodeTour.

In contrast, participants ct2 and ct3 quickly skimmed through the CodeTour annotations at the start of the task. Participant ct2 never returned to either of the tours, but participant ct3 returned to either of both code tours four times throughout the course of their time working on the task. Each repeat visit to the tour was very brief though. Figure 4 shows a timeline view of ct3's activity over a roughly 20 minute period of time while working on the task.

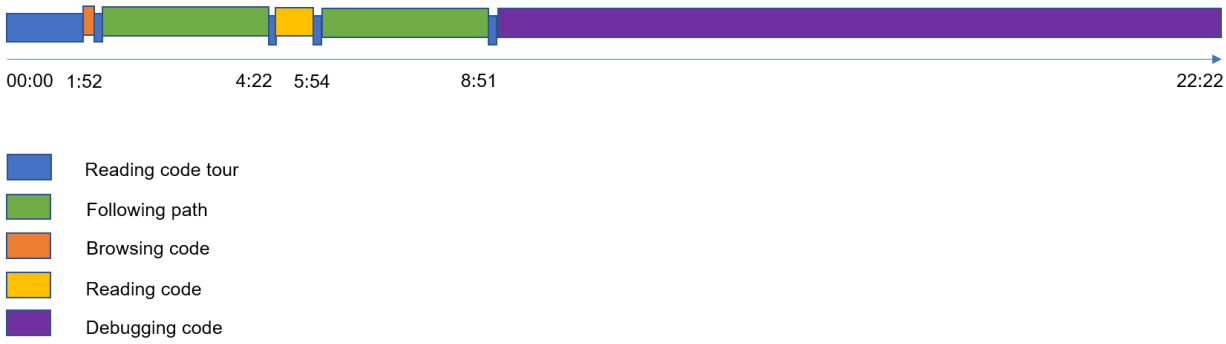

Figure 4 A timeline showing a sequence of activities that participant ct3 performed while working on Task 1.

In the timeline view, coloured blocks represent distinct activities that the participant performed. The length of the block indicates how long the participant performed that activity. Transitions between activities are indicated by a change in colour and a change in position of the activity block. The key underneath the timeline describes the activity that is represented by each coloured block. Most are self-explanatory with the exception of "Following path." This activity represents any time a participant uses any of the code navigation tools inside the editor, such as "goto definition", "find all references", "goto implementation" two or more times in succession. Effectively the participant is linking together and traversing two or more locations in source code, following a path from one location to another.

The timeline shows that the participant spent approximately two minutes reading the code tour then returned to it briefly four times (the very short blue blocks that intersperse other longer periods of activity).

It would appear that this participant used the CodeTour to identify points of interest inside the codebase to navigate to. He rarely referred to the file structure of the codebase and instead selected a particular step in a code tour to navigate to that corresponding location in code.

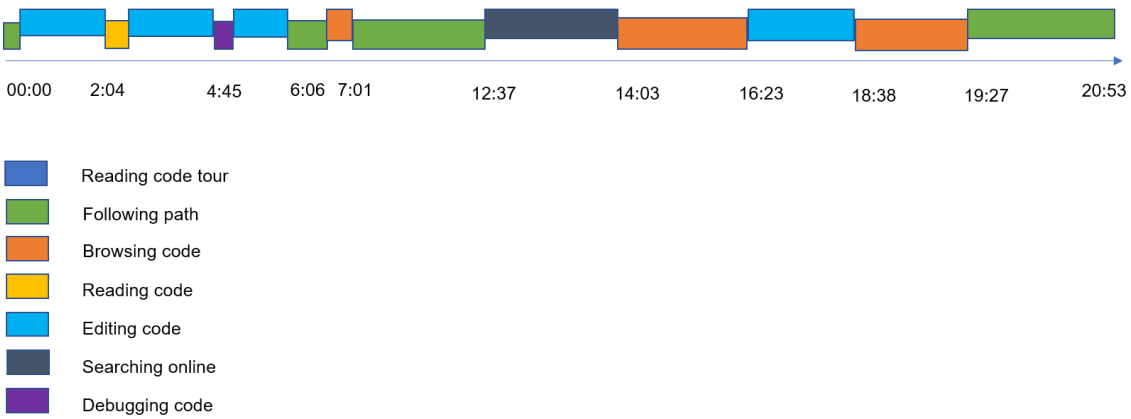

Figure 5 A timeline showing a sequence of activities that participant noct4 performed while working on Task 2.

In contrast, the timeline view shown in Figure 5 shows that participant noct4 spent a lot of time jumping between different activities while working on task 2. Noct4 had almost completed the task but was missing one key line of code that was preventing them from completing the task successfully.

After 14:03, the participant spent the remainder of the time browsing code to search for some clues that might help them complete the task. They ended up following multiple paths and browsing multiple files without finding anything that helped them.

Identifying cause of bug in task 1 – noct group

In the non-CodeTour group, participants browsed and searched through more files than the CodeTour group. None of the non CodeTour participants were able to identify the cause of the bug.

For example, participant noct2 was observed continuously switching between different files to try to identify the location of the bug. At first, he relied on the names of files to try to identify potential sources of the bug. He initially believed that the file EraserPaint.java must contain the code responsible for the bug and the undo functionality ("I am not sure how this works, the issue must be with EraserPaint"). Other participants behaved similarly, looking at files like EraserPaint.java and PencilPaint.java, in contrast to code tour participants who did not look at these files at all (see Table 2).

Implementing line tool in task 2 – ct group

For the second task, participants were required to implement a new class that would be responsible for drawing straight lines and then write code that would wire the instantiation of that class to the appropriate UI control. Participants could reference the implementation of the Pencil tool while implementing the line drawing tool. But to figure out how to wire that new class up to the UI control they would have to discover how paint objects are constructed. As in task 1, there were intermediate classes involved, with the PaintObject and PaintObjectConstructor classes being responsible for creating instances of the pencil and line drawing tools.

Table 3 shows the average amount of time each participant spent in each file and the number of times they opened that file while working on task 2. We don't see a major difference between the two groups, but the file PaintWindow.java was crucial, and this was where many participants got lost.

Task 2: Average time and number of times active in each file per participant				
	Did not use code tour		**Used code tour**	
Time on task	21:25		23:19	
	Time in file	**Active**	**Time in file**	**Active**
Actions.java	03:13	4	04:37	5
EraserPaint.java	00:10	1	00:17	1
PaintCanvas.java	01:54	1	00:43	2
PaintObject.java	00:07	1	00:14	1
PaintObjectConstructor.java	00:54	1	01:59	2
PaintObjectConstructorListener.java	00:42	1	00:01	1
PaintWindow.java (Useful file)	01:53	3	05:28	5
PencilPaint.java (Useful file)	02:28	4	02:54	6

Table 3 Number of times that a file was active in the editor while participants worked on task 2.

Two participants, ct5 and noct8 were the only two participants able to complete both pieces of functionality for this task. Most participants were able to implement a class that would be responsible for drawing straight lines simply by referring to the existing implementation of the PencilPaint class.

Participant noct7 created a LinePaint file after inspecting the pencilAction.java file. However, after meandering between the different files, he gave up on the task ("This is as far as I'm going to get").

Discussion

Our results suggest ways that tools like CodeTour may influence how developers become familiar with an unfamiliar code base.

We attempted to validate the following hypotheses:

H1: Code tours will provide a useful starting point for people orienting themselves to a new code base

H2: Code tours will encourage developers orienting to a new code base to explore more of the code base than they would have done otherwise

Our observations lead us to believe that CodeTour could provide a useful starting point is by making it clearer which files are relevant to a task and which are not. During the study, we observed that participants in the code tour group did not look at PencilPaint.java or EraserPaint.java during task 1 while those in the non code tour group did. Indeed, we observed non code tour participants using cues like file names as their way to forage for potentially relevant code. In such cases, ambiguity and lack of clarity in names can lead to time wasted browsing irrelevant code.

It is interesting that even in such a small codebase we observed these differences. With only eight files in the whole code base, it is difficult to spend a significant amount of time browsing irrelevant code. However, in a larger, more realistic code base with many tens or hundreds of files, it is conceivable that a developer could spend a large amount of time browsing irrelevant code as they forage for code that could help them complete their task. In these situations, CodeTour could potentially help by identifying files that are relevant.

Thus we believe that CodeTour does have some promise in helping developers orient to a new code base but this is still to be verified.

Similarly, another interesting observation was the way that some CodeTour participants used the CodeTour to navigate to interesting locations in code. We observed participants using the tours to navigate to locations in code that were associated with particular steps in a tour, rather than by file

name. We believe that these participants began to use the code tour as their primary way of referencing code, rather than referencing the code by the file that it was contained in.

This does suggest that a well-written code tour could provide a useful starting point for developers orienting to a new code base but it clearly depends on the availability and quality of the code tour.

However, both these findings do suggest that our second hypothesis would be invalidated. We saw that code tour participants activated fewer files than the non code tour participants, for example. It is possible that the code tour could actually discourage developers from exploring parts of the code that aren't covered by a code tour, perhaps thinking that since they are not covered by a code tour that they are not interesting or relevant. While reducing the amount of time spent browsing irrelevant code is worthwhile, reducing opportunities for serendipitous discoveries of useful code simply due to it not being contained in a code tour would be potentially an even worse outcome.

The scenario that our participants were placed in also had a bearing on our observations. We placed participants in the scenario of maintaining a legacy application. In such a situation it might be the case that developers aren't as motivated to spend time learning the code base. Perhaps all they care about is learning enough to be able to complete their task (fix a bug, add a new feature) and then move on. We believe that we observed some of this behaviour during our study as we saw that most CodeTour participants didn't initially care to learn the codebase. Two CodeTour participants were only motivated to fix the bug and move to the next task. They considered the cost of reading through a code tour to the benefits that they expected to gain. They weren't as motivated to learn about the code base since they thought that they just needed to do enough to fix the bug.

Another participant though suggested that they are always conscious about the risk of introducing a new bug while fixing an existing bug. They carefully read and understood the CodeTour before proceeding but it is clear that other developers might look at the cost-benefit analysis differently. If the developer is addressing what they consider to be a trivial bug in a legacy code base, the cost to learn the codebase could be higher than the effort involved to fix the bug.

One factor which would affect how developers determine if the cost of browsing a code tour is worth it is their reaction to a largely text based medium. One CodeTour participant said that the CodeTour was overwhelming at first: "*Coming into the first task I think I was confused with all the source code, you know the tool tips given etc. I got confused. Instead of debugging, I spent a lot of time in understanding the structure of the code.*". They felt that they might have had a better time stepping through the code in the debugger to build up an understanding of how the code works, rather than reading the steps in each code tour (what they referred to as the tool tips).

On reflection, although there was not a clear difference in success rates between those participants that had access to the code tours and those that did not, participants that used the code tours suggested that they were useful in helping them get started and understanding the flows through the code. For example, after using the code tours, participants were asked for their thoughts:

- ct1: "*Code tours were really useful. Normally for this application there are many classes, so it guided me… to a particular spot on the code and it was like properly displayed in a great UI form, and I was able to figure out what this functionality is for…Because the application which I work on are performance dependent like even a small code change [if] it goes wrong it can be disastrous. So, like, I have to carefully see all the flows which are possible… the CodeTour was very useful because it redirected me to the particular function it was trying to describe and it was presented in a more UI friendly way, so it was very readable and I was able to understand everything.*"
- ct2: "*Without the code tour that would have taken me more time to find where is the start point.*"
- ct3: "*CodeTour was really helpful, description was really helpful. The navigation was really helpful. After the first task I came to realize CodeTour really helped.*"
- ct5: "*It instills a level of confidence that somebody has thought this through."*

In conclusion, we believe that our study invites future exploration into how developers learn, navigate, and ask questions about code.

Limitations and threats to validity

Given the small number of participants in the study, our results are not statistically significant and so we cannot make any claims about the impact of CodeTour.

Furthermore, the scenario in which we observed participants was limited to that of spending a short amount of time addressing a bug and implementing a new feature in a legacy codebase. We did not observe participants in a scenario where they were working with a new codebase over an extended period of time.

Considering the codebase is small and the study length was one hour, on average all participants spent a roughly equal amount of time in the most relevant classes, but a typical legacy codebase would be longer and include code written by many different developers, and so developers would find it more difficult to find the most relevant classes.

Another limitation is that the study participants were all male. We know from the work of (Burnett et al, 2016) that there are differences between males and females in how they approach tasks like these. We have not been able to observe how female developers would use CodeTour to approach these same tasks.

Many developers are able to ask for help from mentors or colleagues when working on tasks like the ones we used in our study. However, to avoid leading or biasing participants we declined to answer any questions about the codebase.

Not all participants were familiar with the debugger in Visual Studio Code and in several instances, we coached participants to operate the debugging controls, which decreased the time available to complete the task.

CodeTour's usefulness is limited by the authors' ability to share helpful annotations, and for this study, we authored and refined two sets of accurate steps, which may not be realistic for every application of the tool.

References

Aghajani, E., Nagy, C., Linares-Vásquez, M., Moreno, L., Bavota, G., Lanza, M., & Shepherd, D. C. (2020). Software documentation. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. https://doi.org/10.1145/3377811.3380405

Aghajani, E., Nagy, C., Vega-Marquez, O. L., Linares-Vasquez, M., Moreno, L., Bavota, G., & Lanza, M. (2019). Software documentation issues unveiled. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/icse.2019.00122

An Overview of the Software Engineering Laboratory [PDF]. (1994). https://ntrs.nasa.gov/api/citations/19950022293/downloads/19950022293.pdf

Begel, A., & Simon, B. (2008). Novice software developers, all over again. *Proceeding of the Fourth International Workshop on Computing Education Research - ICER '08*. https://doi.org/10.1145/1404520.1404522

Begel, A., Khoo, Y. P., and Zimmermann, T. (2010). Codebook: discovering and exploiting relationships in software repositories. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. https://doi.org/10.1145/1806799.1806821

Burnett, M., Peters, A., Hill, C., and Elarief, N. (2016). Finding Gender-Inclusiveness Software Issues with GenderMag: A Field Investigation. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. Association for Computing Machinery, New York, NY, USA, 2586–2598. https://doi.org/10.1145/2858036.2858274

Dagenais, B., Ossher, H., Bellamy, R. K. E., Robillard, M. P., and de Vries, J. P. (2010). Moving into a new software project landscape. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. https://doi.org/10.1145/1806799.1806842

de Janasz, S. C., Sullivan, S. E., & Whiting, V. (2003). Mentor networks and Career Success: Lessons for Turbulent Times. *Academy of Management Perspectives*, *17*(4), 78–91. https://doi.org/10.5465/ame.2003.11851850

DeLine, R., Venolia, G., and Rowan, K. (2010). Software development with code maps. *Communications of the ACM*, *53* (8), 48–54. https://doi.org/10.1145/1787234.1787250

DeLine, R., Czerwinski, M., and Robertson, G. (2005). Easing program comprehension by sharing navigation data. *2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. https://doi.org/10.1109/VLHCC.2005.32

Gordon, M., and Guo, P. J. (2015). Codepourri: Creating visual coding tutorials using a volunteer crowd of learners. *2015 IEEE Symposium on Visual Languages and Human-Centric Computing.* https://doi.org/10.1109/VLHCC.2015.7357193

Horvath, A., Liu, M. X., Hendriksen, R., Shannon, C., Paterson, E., Jawad, K., Macvean, A., and Myers, B. A. (2022). Understanding How Programmers Can Use Annotations on Documentation. *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. https://doi.org/10.1145/3491102.3502095

Ko, A. J., DeLine, R., and Venolia, G. (2007). Information Needs in Collocated Software Development Teams. *Proceedings of the 29th international conference on Software Engineering (ICSE '07)*. https://doi.org/10.1109/ICSE.2007.45

Ko, A. J., and Myers, B. A. (2008). Source-Level Debugging with the Whyline. *Proceedings of the 2008 International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE '08)*. Association for Computing Machinery, New York, NY, USA, 69–72. https://doi.org/10.1145/1370114.1370132

Loksa, D., Ko, A. J., Jernigan, W., Oleson, A., Mendez, C. J. and Burnett, M. M. (2016). Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. Association for Computing Machinery, New York, NY, USA, 1449–1461. https://doi.org/10.1145/2858036.2858252

Maalej, W., & Robillard, M. P. (2013). Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering*, *39*(9), 1264–1282. https://doi.org/10.1109/tse.2013.12

Oney, S., Brooks, C., and Resnick, P. (2018). Creating Guided Code Explanations with chat.codes. *Proceedings of the ACM Human-Computer Interaction, 2*(CSCW), 1–20. https://doi.org/10.1145/3274400

Robillard, M. P., & DeLine, R. (2010). A field study of API learning obstacles. *Empirical Software Engineering*, *16*(6), 703–732. https://doi.org/10.1007/s10664-010-9150-8

Sillito, J., Murphy, G. C., & De Volder, K. (2008). Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, *34*(4), 434–451. https://doi.org/10.1109/tse.2008.26

Stettina, C. J., and Heijstek, W. (2011). Necessary and neglected? an empirical study of internal documentation in agile software development teams. *Proceedings of the 29th ACM international conference on Design of communication (SIGDOC '11)*. https://doi.org/10.1145/2038476.2038509