In P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds). Proc. PPIG 17 Pages 92 - 105

Theoretical Considerations on Navigating Codespace with Spatial Cognition

Anthony Cox[1], Maryanne Fisher[2], and Philip O'Brien[1]

[1] Computer Science, Dalhousie University, Halifax, NS, CANADA
[2] Psychology, Saint Mary's University, Halifax, NS, CANADA

Abstract. Spatial cognition has been well examined using various psychological perspectives. Here we suggest that this previous research can be utilized to provide insight into source-code navigation and program comprehension. In our model, code represents an abstract space that must be navigated using the same cognitive strategies as for natural environments. Thus, when navigating 'codespace' computer programmers face many of the same challenges as people navigating within the real world, and consequently, will likely rely on similar skills and strategies. In support of this observation, we explore from a theoretical perspective the use of spatial cognition during program comprehension. Examination reveals that research in spatial cognition provides, albeit currently unproven, explanations for programmer behaviours during program comprehension activities. To validate our model, we suggest a preliminary experiment to explore the existence of codespace.

1 Introduction

Program comprehension, reading a map, and driving an automobile are tasks that, at first, appear to have little in common. However, it is highly unlikely that we have developed a highly specialized skill set for each of these tasks. Instead, as a result of evolutionary pressures, we have developed a small set of widely adaptable, general purpose cognitive skills and structures [1]. When given a new activity that requires a cognitively complex skill set, the needed skills are generated by adapting and combining existing skills [2]. Furthermore, mechanisms and structures that have proved useful in one domain will be used in similar domains and will be changed only when driven by the need for performance improvement [2]. Hence, an important question to ask is – To what other activities does program comprehension exhibit similarities?

Our focus on program comprehension stems from its importance with respect to computer programming. Computer programs are formal specifications for describing a set of actions that one wishes a computer to perform. These specifications are written in an artificial language known as a computer programming language. The text of the program is known as the program's *source-code*. While source-code can be generated by anyone familiar with the programming language's syntax, a program that accomplishes a desired task can only be created by a programmer that comprehends or understands the semantics (meaning) of the language and the program's source-code. That is, almost all software maintenance activities require a programmer to comprehend the software's

source-code, making program comprehension one of the key human elements of computing.

While there are many models and corresponding definitions for the term *program comprehension*, for the purposes of this paper, we adopt the definition of Biggerstaff, Mitbander and Webster [3]:

> "A person understands a program when able to explain the program, its structure, its behaviour, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source-code of the program."

Using this definition, program comprehension is the process of generating a cognitive map from the program source-code to a set of application domain concepts. The resulting cognitive map is referred to as a *code-to-concept map*. An example of a typical application domain is the banking domain and example concepts within this domain include accounts, assets, deposits and compound interest calculations. The set of application domain concepts is limited only by one's ability to describe, communicate and reason about them.

As the need for program comprehension skills has developed only recently, in evolutionary terms, these skills must result from the 'new' application of existing skills. We postulate that some of the skills used during program comprehension overlap significantly with those used for spatial cognition. In other words, we equate the navigation of source-code when building a code-to-concept map, with 'wayfinding' when exploring a real-world geographic space. This equality identifies a new component of program comprehension that explains documented phenomena and generates new research opportunities.

Support for this view is provided by Green [4], who postulates that "mental representations of programs seem to use spatial imagery where possible." Evidence also suggests that, when possible, programmers use spatial imagery as a coding for program concepts [5]. Green and Navarro found that program comprehension was facilitated when related program concepts occurred in close physical proximity [5].

Douce *et al.* [6] suggest that comprehension and maintenance substantially use programmer's spatial abilities and that the source-file can be viewed as "program space." Singer *et al.* [7] have developed a tool called 'NavTracks' to support navigation in "software space." We support the idea that source-code creates an abstract space and examine it more fully in this paper. Using the literature on spatial cognition, we explore source-code navigation during program comprehension in order to provide new insights into some of the cognitive processes that occur.

Sections 2 and 3 provide overviews of program comprehension and spatial cognition, respectively. In Section 4, the structure of source-code is examined and its similarities to the physical world are identified. Using these similarities, source-code navigation is examined in Section 5 to relate previous research on program comprehension and spatial cognition. In Section 6, we present a preliminary experiment to investigate our hypothesis that source-code and the physical world are navigated using the same cognitive skills.

2 Program Comprehension

Program comprehension begins with source-code and ends with a mental model describing the code's purpose, operation, and abstractions. To construct this model, programmers must examine the text of the source-code using some form of strategy.

Brooks [8] hypothesised that maintainers use a top-down hypothesis driven model of comprehension. Beginning with a primary hypothesis, programmers examine source-code to confirm and refine this hypothesis, generating new supplementary hypotheses as part of the process. Soloway and Erlich [9] suggest that a top-down approach is used by maintainers that are familiar with the source-code. Using "rules of programming discourse," programmers match code fragments to a suspected "program plan" that generalizes similar and familiar programs. In both these models, programmers begin at a high level of abstraction and move towards a lower level.

Pennington [10] finds that maintainers use a bottom-up model of comprehension and begin by forming a control flow "program model" based on the syntactic elements of the source-code. A "situation model" is then constructed to describe the program in terms of semantic objects and concepts. In bottom-up comprehension, programmers begin at a low level of abstraction and move towards a higher level.

To unify the bottom-up and top-down comprehension approaches, an integrated comprehension model was developed by von Mayrhauser and Vans [11]. In the integrated model, maintainers opportunistically shift between top-down and bottom-up approaches. Generally, programmers begin using a top-down approach if they have sufficient experience to generate high-level hypotheses or they begin using a bottom-up approach when they lack this experience. A shift between approaches occurs when either they lack the experience to form additional hypotheses (top-down to bottom-up) or they encounter a recognizable element (bottom-up to top-down). Switches continue to occur as opportunities present themselves.

These models suggest that programmers use multiple strategies during program comprehension, depending upon a variety of contextual factors. Some of these factors, such as programmer experience [10], have been identified, while others remain unknown. However, independent of strategy, these models are based on the examination of source-code. As documented by Mosemann and Wiedenbeck [12] programmers do not examine a file sequentially. Thus, as part of program comprehension, programmers must develop some mental representation of the source-code so that they are aware of where the features they are examining are located within the code.

3 Spatial Cognition

Just as individuals vary in their use of a program comprehension strategy, they similarly vary in their performance of navigational tasks. As will be demonstrated in this section, the literature on spatial cognition indicates that individual, task and situational differences all affect the choice of skills used for spatial cognition.

Spatial cognition has been defined by Downs and Stea [13] as:

> "A process composed of a series of psychological transformations by which an individual acquires, stores, recalls, and decodes information about the relative

> locations and attributes of the phenomena in their everyday spatial environment."

Spatial cognition is composed of several elements: landmarks, route maps and survey maps. Landmarks are identifiable environmental markers associated with specific geographic locations [2]. Route maps are sequences of instructions, often involving landmarks, that describe at a personal level how to get from one location to another [14]. Survey maps are similar to topological maps and describe the spatial layout of the environment as opposed to reflecting a specific navigational task [14].

It is well accepted in the spatial cognition literature that Piaget's model for children is correct and that we develop and learn to use these concepts in the order they are listed [15]. However, as adults, when placed in a new and unfamiliar environment, we do not learn about the environment in the same order. Additionally, as part of spatial cognition, individuals must also determine their own location. Self location depends on ones *sense of direction* for which we use the definition of Kozlowski and Bryant [16]:

> "the awareness of location or orientation which specifies where the participant is when he or she moves around the environment."

Moeser found that after two years in the same building, nurses had not developed survey maps of a hospital and used other strategies to navigate [17]. Appleyard [14] suggests that about 75% of the population use route maps and the remainder survey maps, but that this choice is mediated by need. Drivers are more likely to use a survey map while bus passengers are more likely to use a route map. It is also suggested that routes and landmarks are inseparable. A landmark is identified when one is needed to associate with an important element of a route, such as a change of direction.

Aginsky *et al*. [18] show that the use of a route oriented or survey oriented strategy is an individual preference and that users of the survey oriented strategy use it before developing a prerequisite route map. However, one is not restricted to using a sole strategy as Kitchin [19] found that one person uses many strategies.

Kato and Takeuchi [20] discovered that, independent of wayfinding ability, individuals are able to use both survey maps and route maps and switch between the two strategies. However, individuals with a better sense of direction made more effective use of each strategy and switched more appropriately. They also suggest that wayfinding is an opportunistic process and that individuals change strategies at different parts of a route. Also, one's choice of strategy is often dictated by the environment. For example, when a lack of sufficient landmarks exists, one adopts a survey strategy.

Thus, spatial cognition is a complex task for which individual differences exist. Of the multiple cognitive skills such as using routes or survey maps, there is considerable variation depending upon the individual, the task and the current situation. As with program comprehension, an individual will often use more than one strategy or skill and will change according to their current needs and situation.

4 Codespace

Experienced programmers are comfortable viewing source-code from many different perspectives [21]. The *structural* view of source-code considers the code's structure –

its layout, indentation, organization into lines and files, use of blank-lines, and so forth. The *syntactic* view organizes source-code with respect to the syntax of the programming language in which the code is written. There are many *semantic* views of source-code, with each view applying the semantics of a specific domain. For example, in the programming domain the source-code is considered with respect to the data structures and algorithms commonly used by programmers. In the business domain, the code is viewed using the terminology of business, such as assets, accounts, and interest calculations.

The views of source-code, structural, syntactic and semantic, are considered as increasing in their complexity. That is, the structural view is considered as the most simplistic, and the various semantic views as the most complex. For the most part, the structural view of source-code has been ignored in the study of program comprehension. It has been shown that the structure of source-code significantly affects its readability and comprehensibility, as it permits the code to be more readily decomposed into syntactic units [22, 23]. However, once a syntactic model has been generated, there is little further reference to the structural model.

We suggest that the structural model is more important than currently believed, as it provides a basis with which programmers can 'move around' the source-code. Our hypothesis is that programmers form a mental model using the structural elements of the source-code and then use this model to navigate the code. As it is likely that source-code navigation will use the same skills as real-world navigation, we refer to this mental model of source-code structure as *codespace*. More formally, we define codespace as an abstract world corresponding to a programmer's view of source-code with respect to the format, layout and organization of objects identified within the code.

In addition to the source-code, there are many other representations for a software system, such as flow graphs, class hierarchies and architecture diagrams. We define codespace with respect to the source-code, as opposed to the entire set of system representations, as these alternative representations can all be considered as abstractions of the source-code. Codespace is only an element of a software system. While abstract concepts are frequently mapped to the source-code that implements them, it is the code that populates codespace and not the abstract concepts. Codespace is populated with source-code constructs, such as type definitions and procedures, not with abstract domain concepts such as bank accounts and withdrawals.

When we move to a new city, our knowledge of how to get around the city increases with practice and experience. Similarly, as programmers become more familiar with code, they should become more adept at navigating codespace. It is likely that as programmers develop more complex views of the source-code, they determine structural positions for any new-found objects, and thus, increase the number of known and meaningful objects in codespace. For example, when viewing code that has not been previously examined, a programmer forms a simple structural model of the code using only the code's physical layout. Then, as the programmer identifies syntactic elements such as subroutines and loops, the positions of these syntactic elements are located in the structural model to increase the number of recognizable objects. Codespace should grow in detail and size as programmers become more familiar with source-code.

Codespace and our physical world have many similarities. These similarities are now presented to provide evidence that the same skills are used to navigate both worlds.

First, both have a physical manifestation that can be viewed and manipulated. Programmers can modify source-code using a tool such as text editor, just as we can modify the physical world using a tool such as a shovel.

Second, both worlds have identifiable constructs (objects). In the physical world, there are rivers, mountains, buildings, roads and so on, while in codespace there are functions, subroutines, loops and blocks (compound statements). In codespace, editors and visualization tools function as eyes, permitting programmers to view the world and identify objects. However, in both worlds, the number of objects that exist covers a larger geographic space than can be simultaneously viewed. Hence, navigation must be performed to explore these worlds and gather information for use in building cognitive models for understanding and comprehension. Just as we would have limited knowledge of the real world if we never left our home, equivalently, programmers will have little knowledge of a software system if they never view more than a single screen of code.

Third, both worlds have the property of direction. In the physical world, we use terms such as 'up,' 'left,' and 'north' to identify direction. In codespace, we can move 'forward' and 'backward' in a source file, and 'left' or 'right' on a specific line in that file.

Cartesian space is generally reduced to one dimensional space in source-code. Programmers tend to assign a linear ordering to code, just as we assign a linear ordering to the words, sentences and paragraphs that make up this paper. The linear nature of code is made explicit in older dialects of BASIC and Fortran where all lines are monotonically numbered. This simplifies the concept of *direction*, since one can only go forwards or backwards in a file. The treatment of codespace as uni-dimensional results from the fact that to improve readability, each line usually contains a single meaningful syntactic element.

Fourth, both worlds can be considered at various levels of abstraction. In the physical world, we can describe a person's office as room 322 in the computer science building of Dalhousie University in the city of Halifax and the country of Canada. Thus, a hierarchy of abstraction levels exists, presented here from most detailed to least detailed. In codespace, one can reference the guard clause in the partitioning loop of the 'quicksort' subroutine in the file 'utilities.c' of the program 'Excel.' Again, a hierarchy of abstraction levels exists. These abstraction hierarchies permit the reduction of large spaces to smaller more manageable ones.

Fifth, there are concepts of distance and layout relating objects in both worlds. Two subroutines can be adjacent, just as two buildings are, while two functions can be 4 screens (of a monitor) apart just as two mountains are 7 kilometers apart. Programmers typically view distance in terms of *lines* and *screens* due to the historical 'transit' techniques – pressing the ↑, ↓, page-up, and page-down keys.

In codespace, there are effectively two transit techniques: scrolling and jumping. Scrolling is the use of the keyboard, window scroll bars, or mouse scroll wheel, to move the cursor forwards and backwards through a source file. Scrolling is the equivalent of 'walking' in codespace since one can watch the 'landscape' pass by during the process. The amount of code that is displayed and the amount of time it takes to display it provides programmers with a sense of distance, similar to walking.

Jumping occurs when the entire viewing window (screen) is rewritten as a result of accessing a search result or a previously marked location. Closing the current file and opening a new file is equivalent to jumping to the start of the new file. Jumping is like instantaneous teleportation, as there is no scaled time lag to provide any sense of distance and no sensory input to indicate direction. We have not yet identified any existing real world transit that displays the same characteristics as jumping.

While codespace and the real world have many similarities, they also have significant differences as the existence of jumping illustrates. These differences do not prevent the same skills from being used in both environments. Instead, the differences suggest that a common subset of skills is used in each world and that this subset is augmented and modified to deal with the unique aspects of each. Cosmides and Tooby [24] support this view in their "swiss army knife hypothesis."

The similarities listed here suggest that both worlds can be navigated using the same cognitive skills. As humans, it is likely that we plot a route to get to our office using the same mechanism as a programmer uses to get to the 'quicksort' subroutine. In the next section, we use research on program comprehension and spatial cognition to identify similarities between navigation in the real world and navigation in codespace.

5 Navigating Codespace

From an evolutionary perspective, humankind has only recently needed skills for performing program comprehension. Hence, the skills we use for this task should have resulted from the adaptation of an existing skill used for some other task. Furthermore, as source-code provides the notational representation for a complex set of domain and programming concepts, the skills needed to manipulate and understand code must be similarly complex and have arisen to manipulate and understand an equally complex environment. For the remainder of this section, we exploit the similarities between source-code and the real world to explore the hypothesis that codespace creates an abstract environment that must be navigated and manipulated using the same cognitive skills as for physical space.

Mosemann and Wiedenbeck consider a programmer's "working through source-code" as source-code navigation [12]. They describe navigation as the path that a programmer takes when performing comprehension. We do not believe that this is an accurate view of navigation, since it equates navigation with a specific task. More accurately, navigation can be considered as the behavioural and cognitive elements used to identify one's position and to move to other identifiable positions [25]. Using the latter definition, we now examine codespace navigation.

5.1 Visual Sub-Systems

Viewing the world is much like program comprehension – we gather data using our visual system and then generate meaning for this data. During program comprehension, we read source-code to gather data, and then map the data to application domain concepts to give it meaning.

Within the human visual system, it has been shown that there are two separate subsystems: the *contour* and the *location* systems [26]. The contour system identifies objects while the location system determines an object's spatial location. Analogously, we can expect to find parallel systems in place for navigating source-code.

In codespace, there are also objects and locations. Objects are meaningful source-code fragments. These fragments can be identified with respect to the syntax of a programming language or with respect to the task that the fragment performs. An example of the former is a block (grouped sequence of statements) in a block-structured language such as C or Pascal and an example of the latter is a group of statements that swap the values of two memory locations. As mentioned, codespace is linearly structured permitting objects to have a location within the single available dimension.

Research on program comprehension suggests that programmers generate meaning for a program by assigning code fragments to domain concepts. This task can be considered as using the contour subsystem. That is, programmers identify meaningful conceptual objects that can be associated with the code.

Current research on program comprehension pays little attention to the equivalent of the location system. There is little examination on the location of these objects, or of their manifestations in source-code. Program comprehension models specify the cognitive structures used to generate meaning for code and ignore the spatial relationships between code fragments and domain concepts. It is likely that this omission is tied to the lack of attention that is given to the structural view of source-code. As well, by ignoring positional relationships the current models of program comprehension can be viewed as incomplete. Addressing this incompleteness can lead to an improved knowledge of the comprehension process and in turn can motivate the production of better techniques, environments, and tools for software maintenance.

5.2 Strategies for Navigation and Comprehension

While we have been considering source-code navigation as an element of program comprehension, there is some benefit in examining the relationship between comprehension strategies and spatial cognition navigation strategies. However, when directly compared, the two areas do not exhibit much similarity or appear to be highly related. The key to achieving a successful comparison is found by using known gender differences in program development and spatial cognition. While program development is not the same as program comprehension, evidence suggests that the two tasks will exhibit similar differences in source-code navigation. This evidence is now presented.

Green, Bellamy and Parker [27] have observed that programmers do not generate code linearly and will jump forwards and backwards in a file when programming. This behaviour suggests the programmers are building and maintaining a 'geographic' model of the code and using this model to navigate during the programming process. The similarity of navigation models for software development and program comprehension, indicates that research on source-code navigation is independent of the task being performed as it is likely the same model is being used. We next provide a brief description of the two main program development models.

To describe program development strategies, we will use the analogy of writing a book. Top-down program development begins at the most abstract level and creates

a skeleton program lacking detail, much like beginning a book by writing the table of contents. Then, detail is added through the implementation of the subroutines that are directly referenced by the initial skeleton. Detail continues to be added through the development of subroutines that are referenced more and more indirectly from the initial skeleton. The adding of detail is similar to developing a chapter by first generating the section headings, then its sub-headings and so on, until the actual text is written as the final step.

Bottom-up development follows a reversed process. The most detailed elements are developed first and then combined by writing successively higher-level subroutines that use the previously developed lower-level subroutines. In a book, this is like writing, in an arbitrary order, a set of paragraphs and then ordering these paragraphs into sub-sections, then sections, and finally into chapters.

It is documented that women tend to program using a bottom-up approach while men tend towards using a top-down approach [28]. For program comprehension tasks we expect a similar difference will exist as comprehension can also be performed using a top-down or bottom-up approach. It is reasonable to expect that, when feasible, women will tend to use a bottom-up comprehension strategy while men will prefer to use a top-down strategy.

It has also been documented that women, more than men, tend to use less abstract, more concrete representations of the environment, such as landmarks. Conversely, men prefer to use more abstract representations such as cardinal directions (e.g. north, south) [29]. When the level of abstraction is considered, it can be seen that bottom-up programming and landmark identification both use a low level of abstraction. Similarly, top-down programming and cardinal direction tend towards a higher level of abstraction.

Postma *et al.* [30] suggest that route-based navigation can be performed at both an abstract or a concrete level, but that the use of survey maps is predominately performed at an abstract level. If women tend towards the less abstract, then they should tend to use a route-based navigation strategy. Lawton [31, 32] confirms this suggestion and found that women were more likely than men to use route-based strategies whereas men were more likely than women to use survey navigation strategies.

Using these observations, it is possible to suggest that since women tend to favour a less abstract, bottom-up programming style and the less abstract route-based navigation strategy, the two can be equated. That is, route-based navigation is similar to bottom-up program comprehension. Similarly, since men tend towards the abstract and use a top-down programming style and survey map navigation strategies, these approaches can also be equated.

Codespace navigation is only a part of program comprehension. As well, it has been suggested that computing is a male dominated domain that forces women to adopt male strategies to be successful [33]. For these reasons, it is possible that measurable sex differences will not be found. In future work we intend to explore these issues.

5.3 Landmarks

When examining source-code, *beacons* are used to identify features and structures (meaningful objects) within the code [8]. Brooks suggests that programmers scan or

search source-code for beacons that confirm or suggest the existence of specific syntactic and semantic objects. This use of beacons suggests that they may have a relationship to landmarks. However, we know of no research that explores the use of beacons for navigation.

We suggest that beacons are distinct to source-code landmarks. Beacons identify the existence of a specific feature. Just as a square-shaped mound indicates the existence of a probable archaeological site, three sequential assignments using three variables indicates a probable swap and consequently, a sorting algorithm. The role of a beacon is the indication of an object's existence.

Landmarks identify a significant positional location in codespace. In a source file, subroutines that serve a similar purpose are often grouped together and preceded by a comment (computer ignored annotation) that identifies the group's purpose. This comment is a landmark that identifies the start of a subroutine group. Landmarks are used during codespace transit as a navigational element.

Although they are distinct from landmarks, beacons are not unrelated. We suggest that beacons are a component of a landmark. With a church, the spire can serve as a beacon to indicate that a church exists and the church can serve as a navigational landmark. In code, the name 'sort' in a subroutine definition acts as a beacon that suggests the existence of a sort subroutine that can be used as a source-code landmark. As they are a component of landmarks, the known gender differences in the ability to recall landmarks should also be exhibited for beacons.

Although unproven, anecdotal evidence suggests that source-code landmarks exist. Programmers, when working with code, do not always search for a construct that they wish to view. Instead, they search for a different construct that is close to the target construct and then scroll to the target. We believe that programmers are using a landmark – the searched for construct – to perform transit. When they cannot accurately describe the location of a target, programmers use a nearby landmark as part of the navigation process.

Subroutines will likely be found to serve as landmarks. We have observed that when asked, colleagues can identify the ordering of subroutines within a familiar file. As well, in some formatting styles, subroutine names are specifically placed so that they can be used as search targets. Thus, programmers develop a locational model of subroutine locations and use this model to navigate. The subroutines that stand-out, for their role, uniqueness of name, or for some other feature, serve as landmarks for navigation.

Landmarks are not just used as search targets. When scrolling through a large file using a scroll bar, it is difficult to scroll to a specific location in a single movement of the slider. However, programmers have no trouble displaying a desired code fragment, as they can slow down the slider movement so that a passing subroutine can be used to determine the distance to the desired fragment. By using landmarks, programmers can scroll forward and backward through a file without getting lost, even when there is no status line to indicate the cursor's current file position.

When using search tools that permit programmers to jump to an arbitrary location in a file, it is documented that programmers can become disoriented [34]. It is likely that disorientation occurs when there is no visible landmark on the screen. Without an

indicator of the current cursor position, programmers must resort to hunting for such an indicator to determine their current position.

6 Verification

To begin the process of verifying our hypothesis, we have designed an experiment that explores the existence of codespace. From Downs and Stea's definition of spatial cognition [13], it can be seen that the acquisition, storage and recall of information about the relative locations and attributes of a set of entities is a fundamental component of spatial cognition.

A very conservative approach would be to experimentally determine the set of entities that form codespace. However, we believe that such conservatism is extreme. It is highly likely that programmers will use domain concepts to partition source-code into meaningful entities. While some of these concepts may come from more abstract, application domains, the majority of entities are highly likely to be formed from the syntactic units of the code's programming language. That is, for an object-oriented programming language such as Java, codespace will contain the definitions and declarations of classes, methods and fields. Thus, our first experiment explores programmers' ability to learn and recall the spatial relationship between procedural abstractions; a key construct in most programming languages.

As many undergraduate programs use Java as the initial programming language, we have selected it as the source language of the experiment. Using a file containing approximately 30 methods, in a single class, participants will be asked to explore the program with the intent of creating a static inter-method call-graph. We chose this task since it encourages programmers to examine the entire file so that a complete graph can be created. Thirty methods should be large enough to be interesting but small enough to fit within the scope of a short experiment. The selected code, a 712 line file, implements a simple checkers game to minimize the effects of the application domain. We hope to avoid distraction by using a domain that is relatively well known to most participants.

In the study, half of the participants will be given a file with line numbers and half will not. Participants will not be directed to use or concentrate on the line numbers. Thus, any knowledge that they display about these line numbers is an indicator that incidental learning has taken place. Incidental learning occurs only when need, opportunity and motivation exists [35]. Consequently, participants will only gain knowledge about line numbers when it is beneficial to do so. In codespace, line numbers provide a fixed measure of distance, much like mileage indicators along the side of a road. If programmers do not use spatial skills when navigating in codespace, there should be no benefit from the existence of a distance scale. We hypothesise that when given line numbers, programmers will be able to more accurately answer questions about the size and locations of methods although they may not be aware of their use of line numbers.

After participants have been given a sufficient length of time to explore the file, they will be asked perform four tasks. In the first task, participants will be asked to order the signatures of all the methods contained in the file. Given an alphabetized list of signatures, participants will number them according to their order within the file. This task examines the ability of programmers to learn and recall the relative positions

of the methods. The task will be scored by assigning each method 1 point for each method that is correctly positioned before or after it. This score will then be compared against a mean random score where methods have half of their preceding and following methods correctly identified.

In the second task, participants will be asked to identify the approximate line number where a method occurred. This task explores the ability of programmers to recall absolute locations with respect to a fixed measurement system. While it is expected that both groups will perform better than random, it is likely that participants who viewed the version with line numbers will perform better than those that did not. This difference should hold for the third and fourth tasks as well. The task will be scored by calculating the number of lines of error between participant responses and the method's actual location. The distances can be compared against a mean random error equal to one half the file's length.

In the third task, participants will be asked to identify the length in lines of some methods. This task examines the ability of programmers to determine a spatial property for a codespace entity. The length of a method identifies its size along the linear dimension measured by line numbers. Size is an absolute property that is formed from the relative positions of the start and end of the method. To evaluate participant responses, indicated method lengths will be compared against the mean length of the methods in the file.

For the fourth task, participants will be given two methods and asked to indicate the number of lines between the two methods. This distance is a relative property formed from the absolute locations of the two methods. As in the second task, we will compare participant responses against a mean random response that is one half of the file's length.

Based on the results of this experiment, we will potentially gain evidence that programmers use syntactic constructs to populate codespace and that codespace displays meaningful spatial properties such as size and distance. We believe that this experiment provides a foundation upon which future, more indepth, research can be based.

7 Conclusion

In this article, we have used a spatial cognition perspective to examine program comprehension. This perspective suggests that source-code navigation is not a trivial activity occuring while we develop complex domain models of a software system. Instead, the perspective suggests that we are developing highly effective route and survey maps to navigate the system's codespace. Without these navigation skills, maintainers would be unable to locate the source-code fragments associated with a domain concept. While the use of program bookmarks and search tools permits programmers to instantly jump to any location in source-code, spatial navigation skills are still required to positionally relate these locations when scrolling.

The similarities between codespace and the physical world suggest that source-code navigation is a missing element of current program comprehension models. This observation suggests that there is a wealth of new research to be performed to explore the navigational skills that programmers use on a daily basis. In this paper, we have

begun this exploration by presenting a preliminary experiment to investigate the existence of codespace. We believe that the use of spatial cognition to explain phenomena in program comprehension will provide insights that are of value to both the fields of psychology and computer science.

References

1. Clark, A.: Microcognition: Philosophy, cognitive science and parallel distributed processing. MIT Press, Cambridge, MA (1989)
2. Chown, E., Kaplan, S., Kortenkamp, D.: Prototypes, location, and associative networks (PLAN): Towards a unified theory of cognitive mapping. Cognitive Science **19** (1995) 1–51
3. Ted Biggerstaff, B.M., Webster, D.: Program understanding and the concept assignment problem. Communications of the ACM **37** (1994) 72–83
4. Green, T.: Cognitive approaches to software comprehension: Results, gaps and limitations. In: Experimental Psychology in Software Comprehension Studies, Limerick, Ireland (1997) Extended abstract of talk.
5. Green, T., Navarro, R.: Programming plans, imagery, and visual programming. In: Human-Computer Interaction: INTERACT-95, London, UK, Chapman and Hall (1995) 139–144
6. Douce, C., Layzell, P., Buckley, J.: Spatial measures of software complexity. In: 11^{th} Annual Workshop of the Psychology of Programming Interest Group, Leeds, UK (1999) 36–45
7. Singer, J., Elves, R., Storey, M.A.: Navtracks demonstration: Supporting navigation in software space. In: IEEE International Workshop on Program Comprehension, St. Louis, MO (2005) To appear.
8. Brooks, R.: Towards a theory of the comprehension of computer programs. International Journal of Man-Machine Studies **18** (1983) 543–554
9. Soloway, E., Erlich, K.: Empirical studies of programming knowledge. IEEE Transactions on Software Engineering **SE-10** (1984) 595–609
10. Pennington, N.: Stimulus structures and mental representations in expert comprehension of computer programs. Cognitive Psychology **19** (1987) 295–341
11. von Mayrhauser, A., Vans, A.M.: Comprehension processes during large scale software maintenance. In: International Conference on Software Engineering, Sorrento, Italy (1994) 39–48
12. Mosemann, R., Wiedenbeck, S.: Navigation and comprehension of programs by novice programmers. In: IEEE International Workshop on Program Comprehension, Toronto, Canada (2001) 79–88
13. Downs, R., Stea, D.: Image and Environment: Cognitive Mapping and Spatial Behaviour. Aldine, Chicago, IL (1973)
14. Appleyard, D.: Why buildings are known. Environment and Behavior **1** (1969) 131–156
15. Piaget, J., Inhelder, B.: The Child's Conception of Space. Norton, New York, NY (1967)
16. Kozlowski, L., Bryant, K.: Sense of direction, spatial orientation, and cognitive maps. Journal of Experimental Psychology: Human Perception and Performance **3** (1977) 590–598
17. Moeser, S.: Cognitive mapping in a complex building. Environment and Behavior **20** (1988) 21–49
18. Aginsky, V., Harris, C., Rensink, R., Beusmans, J.: Two strategies for learning a route in a driving simulator. Journal of Environmental Psychology **17** (1997) 317–331
19. Kitchin, R.: Exploring spatial thought. Environment and Behavior **29** (1997) 123–156
20. Kato, Y., Takeuchi, Y.: Individual differences in wayfinding strategies. Journal of Environmental Psychology **23** (2003) 171–188

21. Shneiderman, B., Mayer, R.: Syntactic/semantic interactions in programmer behaviour: A model and experimental results. International Journal of Computer and Information Sciences **8** (1979) 219–238
22. Miara, R., Musselman, J., Navarro, J., Shneiderman, B.: Program indentation and comprehensibility. Communications of the ACM **26** (1983) 861–867
23. Baecker, R., Marcus, A.: Human Factors and Typography for More Readable Programs. Addison-Wesley, Reading, MA (1990)
24. Cosmides, L., Tooby, J.: Cognitive adaptations for social exchange. In: The Adapted Mind. Oxford University Press, Oxford, UK (1992) 163–228
25. Passini, R.: Spatial representations, a way-finding perspective. Journal of Environmental Psychology **4** (1984) 153–164
26. Kaplan, S., Kaplan, R.: Cognition and Environment. Praeger, New York, NY (1982) Republished, 1989, Ulrich's, Ann Arbor, MI.
27. Green, T., Bellamy, R., Parker, J.: Parsing and gnisrap: A model of device use. In: Empirical Studies of Programmers: 2^{nd} Workshop. (1987) 132–146
28. Turkle, S., Papert, S.: Epistemological pluralism: Styles and voices within the computer culture. Signs **16** (1990) 128–157
29. Saucier, D., Bowman, M., Elias, L.: Sex differences in the effect of articulatory or spatial dual-task interference during navigation. Brain and Cognition **53** (2003) 346–350
30. Postma, A., Jager, G., Kessels, R., Koppeschaar, H., van Honk, J.: Sex differences for selective forms of spatial memory. Brain and Cognition **54** (2003) 24–34
31. Lawton, C.: Gender differences in way-finding strategies: Relationship to spatial ability and spatial anxiety. Sex Roles **30** (1994) 765–779
32. Lawton, C.: Strategies for indoor way-finding: The role of orientation. Journal of Environmental Psychology **16** (1996) 137–145
33. Cooper, J., Weaver, K.: Gender and Computers: Understanding the Digital Divide. Lawrence Erlbaum Associates, Mahwah, NJ (2003)
34. Janzen, D., De Volder, K.: Navigating and querying code without getting lost. In: International Conference on Aspect-Oriented Software Design, Boston, MA (2003) 178–187
35. Marsick, V., Watkins, K.: Informal and incidental learning. New Directions for Adult and Continuing Education (2001) 25–34