On the Use of Visual Clustering to Identify Landmarks in Code Navigation

Sandeep Reddivari  
School of Computing  
University of North Florida  
Jacksonville, Florida, USA 32224  
sandeep.reddivari@unf.edu

Mahesh Kotapalli  
School of Computing  
University of North Florida  
Jacksonville, Florida, USA 32224  
n00970423@ospreys.unf.edu

Abstract—Recovering the legibility features is key to reverse engineering as the legible software systems can ease developer’s code navigation and comprehension. Landmarks are important legibility features that developers use as reference points. In this paper, we leverage visual clustering to explore how landmarks can be identified via static dependencies. Besides organizing software entities with coherent patterns, visual clustering offers additional insights by rigorously rendering a holistic picture of the code base to the two-dimensional space. We contribute a couple of heuristics based on the cluster layout to identify the landmark files. Our visual exploration of Eclipse Mylyn open source Java project reveals developer’s reliance on the landmarks during code navigation and shows the promise of using static dependencies to uncover the landmarks in the software space.

Keywords—software clustering; program comprehension; code navigation; visual clustering; software exploration; layout-based clustering; software visualization, static dependencies;

I. INTRODUCTION

Code navigation [1]–[5] is concerned with the strategies developers use to find relevant areas of concern in code. Such a process is central to program comprehension and instrumental to software maintenance. Despite the support offered in contemporary SDEs (software development environments), code navigation remains a time-consuming and difficult task. For example, when using Eclipse to perform software maintenance activities, 35% of developer time was spent in navigating through the code [3], and 76% of all navigations referred back to locations that had already been visited [6]. Thus, a major challenge is to understand the fundamental mechanisms underlying developers’ code navigation behaviors and further develop tool support to augment their information seeking skills in software maintenance. As the need for navigating around the code base emerges only within the past few decades, in evolutionary terms, developer’s code navigation skills must result from adapting and combining existing ones [7]. In fact, if a software system is conceived of as an information space (software space [2]), then the general-purpose wayfinding principles are believed to be applicable to program comprehension which requires a cognitively complex skill set [7]–[10].

The foundation for wayfinding research in large-scale geographic environments was laid out by Lynch [11], where he developed a theory of city planning and urban design. Moonen [10] extended Lynch’s theory and showed the significant impact of legibility on software maintainability. Legibility, in this context, is referred to as the ease with which parts of the software space can be recognized. A crucial legibility feature is landmark—a prominent environmental reference point that gives a sense of location and orientation [10], [11]. From the navigation point of view, a high proportion of revisits (e.g., [2], [3], [6]) provides the initial evidence that landmarks exist.

We argue that identifying landmarks is particularly relevant to reverse engineering because, by recovering the essential legibility features, not only can programs be understood at a level of reduced complexity, but software evolution can be tackled in a well organized manner. The need of automatically organizing a large software system’s structure into smaller, more manageable subsystems has given rise to the research area of software clustering. Clustering source code artifacts has contributed to a variety of maintenance tasks, such as remodularization [12], program comprehension [13], architecture recovery [14], concept location [15], and so on.

Despite its key role in reverse engineering, clustering has not been exploited to identify landmarks so as to directly aid navigation. Addressing this knowledge
gap is of special interest because developer's navigation tends to stick to parts of the code [2] and clustering can help discover these parts by organizing them with coherent patterns. Therefore, our central hypothesis is that the synthesis of clustering and legibility features like landmarks offers valuable support for navigating around the software space.

To test the hypothesis, we carry out a visual exploration of the Eclipse Mylyn open-source Java project by focusing on the landmarks in the software space. Our aim is to explore the relationships between developer’s actual navigation and the code base topology defined by static dependencies (e.g., calls, inherits, etc.). As a technological basis for modeling software topology, we leverage visual clustering [16] whose output’s layout provides remarkable insights into the system structure. We further contribute a couple of heuristics to identify the plausible landmarks of the software space. The visual clustering results are then compared with developers’ navigation data. These comparisons help assess to what extent static-dependency-based visual clustering facilitates code navigation.

Our exploratory study investigates the central hypothesis by conducting a quantitative analysis of over 800 maintenance sessions to measure the influence of the landmark on developer’s navigation behavior. The results confirm the existence of landmarks in the software space, reveal the developers’ reliance on the landmarks during code navigation, show the promise of using static dependencies to uncover the landmarks, and open up new research avenues in reverse engineering and program comprehension.

The contributions of our work lie in the characterization of landmarks based on the navigational data and the development of heuristics for recognizing the landmarks based on visual clustering. The static-dependency based landmarks landmark discovered in our study can shed light on the principled ways to increase practical support for code navigation. In what follows, we provide the context of our research in Section II. We then detail our research methodology in Section III. Sections IV describes our quantitative analysis. The implications of our work are discussed in Section V, and finally, Section VI concludes the paper.

II. BACKGROUND AND RELATED WORK
A. Software Clustering and Visual Clustering

Software clustering techniques are employed commonly to aid the reverse engineering process. These techniques attempt to automatically discover natural groupings of large numbers of software entities, such as the set of all files comprising a software system. Their goal is to decompose the problem of understanding a large piece of software into smaller and easier-to-manage subproblems, thus reducing the effort involved in recovering the system structure.

A diverse range of software clustering approaches has been presented in the literature. These approaches differ in the underlying mechanisms, e.g., deducing clusters from file and procedure names [12], computing a measure of structural similarity between software entities [14], minimizing the information loss during the grouping of entities [17], or treating clustering as an optimization problem [18].

Specifically related to our work are source code clustering techniques that facilitate program comprehension and identification of locations relevant to a software change. We classify these techniques into three categories:

1) Pattern-based clustering forms clusters based on the familiar subsystem structures that frequently appear in manual decompositions of large-scale software systems [13];

2) Evolutionary clustering mines the co-change relationships from the software revision repositories [19]–[21]; and

3) Collaborative clustering exploits project teammates’ navigation history to group and recommend software entities [1], [2], [22].

In most approaches, clustering results are presented using flat lists [22] or hierarchical dendrograms [14]. When the clusterings are depicted [18], the visualizations are rather illustrative than schematically rigorous. Visual clustering advances the literature by attempting to ensure the drawn layout fulfills the desired clustering properties [16].

In particular, the software entities to be clustered are represented as nodes of a dependency graph. A visual clustering algorithm then maps each node to a position in the two dimensional space. The layout can be computed by forcedirected graph drawing, in which an energy model encodes the layout goal. For clustering, this means to produce layouts that provide separation of cohesive subgraphs and interpretable distances [23]. Noack [16] thoroughly investigates the existing visual clustering methods, among which the LinLog energy model naturally reveals software clusters and has been successfully applied to address a number of software engineering problems [23]–[27]. We therefore adopt this
B. Spatial Exploration and Software Legibility

The spatial cognition aspect of program comprehension is evident, e.g., Green and Navarro find that, when possible, developers use spatial imagery as a coding for program concepts [9]. Cox et al. provide the theoretical underpinning that suggests developers evolve spatial exploration and wayfinding skills to discover the software landscape [7]. In this way, a software system is considered as an information space through which a developer has to navigate [2]. The research in spatial cognition and wayfinding was pioneered by Lynch’s seminal work [11], where he studied people’s organization of the spatial information about the urban environments. An original and vital finding is the identification of five principal legibility features which people use to build a mental model of a city [11]

- **Landmarks**: Prominent (static) features in a city (e.g., monuments and shop-fronts) that are used as reference points by the observer.
- **Paths**: Streets or footpaths that allow the observer to travel through the city.
- **Nodes**: Important points of interest along paths, e.g., street intersections, bridges, or town squares.
- **Districts**: Areas (e.g., shopping or residential) sharing a common property that can be viewed as a single entity.
- **Edges**: Boundaries to areas that form a physical barrier to travelers; examples include rivers and major roads.

These structural elements are useful for effectively recovering information from a complex environment and for directly guiding the creation of a mental map detailing the spatial knowledge about that environment. Studies show that the more legible an environment is, the easier it is for people to navigate around, recognize a location, and determine a route [11], [28].

Building on Lynch’s work, Moonen defines legibility of software as “the ease with which its parts may be recognized and can be organized into a coherent pattern” [10]. Recovering and improving the legibility features are crucial to reverse engineering and reengineering because legible systems are more memorable and generate stronger mental models, which makes these software systems easier to comprehend and maintain [10]. Extending beyond Lynch’s work, Moonen emphasizes task-relevant legibility features which are likely to greatly help fulfill the developer tasks at hand. For the task of spatial exploration, people make considerable use of landmarks to gain a sense of location and bearing [11].

C. Landmarks in the Software Space

The existence of prominent source code entities has been demonstrated through empirical investigations and exploited in several approaches, though these entities are not termed “landmarks” in all the studies. We distinguish two types of evidence presented in support of the notion of landmark which we use in a similar sense as [10]: the code base itself and developer’s navigation in the code base.

Earlier program comprehension theory suggested the idea of “beacons” as stereotypical code snippets [29]. The beacons are prominent programming features implying a specific functionality, e.g., a variable swap implies a sorting function. Further work showed the existence of beacons and suggested that they could help programmers to quickly identify common functions [30]. Landmarks have also manifested themselves as the entities with a high connectivity in the system structure. In particular, a “dominator node” has paths to all the nodes in a subgraph (subsystem) [13], whereas an “omnipresent object” is connected to a large number of subsystems [31]. Due to the critical role of the structural landmarks, they have received consistent attention in the software clustering literature. More recently, Lawrance et al. recognized the “signposts” in the software space and used words (both identifiers and comments) in the source code to model these proximal cues [32]. The lexical landmarks are shown to guide the debugging process [33].

Studies of software developer’s information seeking behavior suggested that the navigation patterns were focused (sticking to one part of the code) and repeated (returning to previously visited code) [2]. NavTracks thus took advantage of the navigation cycles to recognize the “central” files that were frequently referenced [2]. The frequent occurrence of revisits was also observed in [3], [6]. To share the navigation data among programmers, Lee and Kang proposed NavClus to group code files based on task-relevant navigation sequences [22]. In an attempt to improve feature location, Walkinshaw et al. exploited “landmarks” explicitly [34]. Landmarks in [34] refer to code entities that are thought to be relevant to the given feature and can be elicited from developer knowledge of the software system. In contrast, we are interested in automated ways to identify the
landmarks in the software space and relate the landmarks to developer’s actual navigation.

III. Research Methodology

The objective of our work is to explore “to what extent can the legibility features identified by visual clustering support developer’s code navigation?” To answer the research question, this section first defines what we mean by navigational “landmarks”. We then describe the research instrumentation by presenting our visual clustering implementation for recognizing the static-dependency-based landmarks in the software space.

A. Working Definition of “Landmarks”

In the context of code navigation, we define “landmarks”, or “navigational landmarks”, to be the programming elements that are referenced frequently by the developer in carrying out a given program comprehension or software change task. In another word, landmarks in our definition hinge on the developer’s actual behavior, influenced collectively by the information and the task environments [5]. This definition is in line with the theoretical consideration which relates code navigation to spatial cognition via the use of landmarks [7].

Our definition is intentionally broad: the navigational landmarks reflect not only the characteristics of the developer’s current task, but also the individual browsing and file access idiosyncrasies (if any) [2]. For example, the landmarks used during a software change session may include source code files that need not to be modified; however, if the developer often references (e.g., views and revisits) certain files, we argue that they are integral to the mental model the developer uses to fulfill the task [35].

Although landmarks can be manually elicited [34] or semi-automatically inferred by tracking interaction histories [2], the results are pointed and partial in that they cover only a portion of the software space, usually a small portion containing frequently co-changed files [19]. To enable a more holistic view of landmarks, we propose to leverage visual clustering to automatically generate a layout-sensible picture of the entire software space.

B. Visual Exploration of Software Clusters

Clustering partitions a set of entities into subsets according to certain properties. If the nodes of a software systems dependency graph are to be clustered, then a software clustering algorithm recovers the system structure based on the dependencies in the graph. Work on visual clustering [16] goes one step further and argues that it is not sufficient to just partition a set into equivalence classes. In addition, visual clustering renders each node into the two-dimensional space and introduces rigor to the renderings. The result of visual clustering is a holistic picture that can effectively convey the landscape of the software system.

Two attributes of the visual clustering result are worth mentioning: layout and distance. The layout of a graph \( G = (N, E) \) is a function \( p \) that maps \( \forall n \in N \) to a position in the two-dimensional space. Layout drawing can be systematized by two components: an energy model that ensures the clustering goal and a force-directed graph drawing algorithm that computes a layout with minimal energy [16]. Formally, let \( U \) be an energy function that assigns each layout \( p \) a real number. The best layout for \( G \) results from \( p_* \) such as \( (p_* \in p) \) and \( \forall \hat{p} \in p: (U(\hat{p}) \geq U(p_*)) \). Among the existing energy models, LinLog is a good approximation of the best graph clustering layout [16] and its implementation is available in [19].

The main advantage of a clustering layout is the intuitive and interpretable distance between the data elements: related elements have close positions and unrelated ones have distant positions [16]. The distance not only indicates a degree of relatedness, but also facilitates the visual examination of the spatial characteristics of the whole system. We extended an existing visual clustering tool [19] [53] by concentrating on the static dependencies in the code base [52]. We call this tool extension SDVisu (static-dependency-based visual clustering).

Fig. 1 illustrates the SDVisu tool that we have developed to assist developers in performing reverse engineering activities. The design intent of SDVisu is to support developers in exploring the static dependencies in the source code via quantitative visualizations. The input of SDVisu is the dependency graph whose nodes are source code files and edges are static dependencies between the files. SDVisu’s current implementation considers three types of dependencies: inherits, calls, and references [52]. The dependency graph spans a nontrivial network, in which many forces take effect simultaneously. This illustrates the global view that we pursue with visual clustering. The SDVisu output in Fig. 1-A shows the landscape of Mylyn’s latest release (http://www.eclipse.org/mylyn). In Fig. 1, software clusters are readily identifiable in the layout as separated groups of highly connected nodes. Currently, SDVisu computes the energy value metric [16] to guide the visual identification of well-formed clusters—such operation is
provided in the control panel as an advanced option (cf. Fig.1-B). In the future, we plan to integrate the orphan adoption technique [36] into SDVisu in order to handle the isolated nodes (“orphans”). Each visual cluster of Fig 1-A groups the source code files into a coherent area. SDVisu employs force-directed drawing layout which balances the forces of all the nodes in a cluster. The size of the node is proportional to its edge degree. The larger the node, the higher connectivity it has in the system structure (cf. Fig.1-A). In SDVisu, a particular color can be selected in order to highlight the nodes with large number of dependencies. A zoom-in feature (cf. Fig.1-D) is also implemented, which depicts in more detail the selected cluster(s) nodes and edge information. SDVisu’s keyword search feature (cf. Fig. 1-C) is an effective way for developers to quickly search for a particular source code file, explore the dependencies, and gain insights [52].

Following the literature on software legibility (cf. Section II-B), each visual cluster of Fig. 1 groups the source code files into a coherent district. To enable the identification of landmarks (i.e., prominent features in a district), we propose two heuristics by leveraging the layout and distance attributes of visual clustering.

- **Centroid-based**: The centroid plays an important role in force-directed drawing as it balances the forces of all the nodes in the cluster [16]. Visually, the centroid serves as a landmark of the district.
- **Size-based**: The size of a node is proportional to its edge degree [16]. The larger the node, the higher connectivity it has in the system structure. Structurally, the large node serves a landmark of the cluster.

As shown in Fig. 1, the heuristics are implemented as options in SDVisu, as a landmark can be both centroid and size-based. It is crucial to note that the landmarks identified by the heuristics in SDVisu are merely static-dependency-based. We next present the quantitative inquiry to explore how the SDVisu landmarks relate to the developer’s navigational landmarks.

### IV. Quantitative Analysis

The goal of our quantitative analysis is to tackle two essential issues: quantitative operationalization of navigational landmarks and quantitative assessment of SDVisu’s effectiveness. To allow for thorough examinations, we take advantage of Mylyn’s detailed interaction traces of over 800 programming sessions. Mylyn is an open-source task management system whose monitoring facility helps record a sequence of events (e.g., edit, view, etc.) a developer performs on the Eclipse SDE [44]. Every programming session is linked with its corresponding task (e.g., a bug report, an enhancement request, etc.) in the issue tracking system. As shown in Table I, we study the latest stable release of Mylyn (version 3.7.1) which contains 2405 Java files. Among
Mylyn’s programming logs, 893 sessions modify the files from the latest version. These sessions are contributed by 31 developers on 751 software change tasks.

### A. Quantifying “Landmarks”

The average number of steps involved in Mylyn sessions (79.8) is considerably greater, which implies that Mylyn’s software change tasks are more difficult and the navigations are more concentrated in that fewer files are visited per session (3.7). The revisit ratio is over 95% (76,079) from the navigational step perspective and is about 90% (3,317) when the file access is concerned. These numbers are significantly greater which shows that revisiting is indispensable during code navigation.

To quantitatively measure the “landmark” files defined in Section III-A, we adopt the navigational cycles observed in Mylyn sessions. A cycle, as defined in [2], is a series of files accesses by the developer, beginning and ending with the same file. For example, if the developer accesses file A, then B, then C, then A, a cycle of ABCA with length \( k = 3 \) is formed. Another relevant concept is window size \( n \) that defines the maximum sequence of \( n \) steps within which cycles should be detected. In an earlier effort, NavTracks detected the cycles whose lengths \( k \leq 3 \) within a \( n \leq 4 \) window and used such navigational cycles to determine file associations [2].

Building on the prior work, we consider a file to be a “landmark” if it forms a cycle of length \( k \leq 3 \) within a \( n \leq 4 \) window of some navigational session. The file-access sequence ABCACEFB for example, although A, B, and C form cycles, only A is regarded as a landmark. The B-cycle goes beyond the window size, whereas the C-cycle is not long enough. Note that the maximum window size \( n = 4 \) is based on the observation that longer navigation paths have a greater potential to contain extraneous files [2]; we observe similar phenomena in the Mylyn data.

Fig. 2 shows the step-wise analyses of revisited and landmark files. Even though the developers revisit a large proportion of Mylyn files (82.86%), less than a half of the revisited files form cycles of length \( k \leq 3 \) within a \( n \leq 4 \) window. According to the manual inspections of randomly selected samples from the landmark files, we confirm the high precision of the cycle-based operationalization. In another word, the landmark files resulted from the cycle analysis are indeed referenced frequently by developers in performing software change tasks. Thus, we take the 38.94% of the code files to be the answer set to assess SDVisu’s effectiveness in extracting landmarks.

| Connect-driven Landmarks: Developer(s) Working on the Same Software Change Task |
|--------------------------------|-----------------|-----------------|----------|
| Same Developer | SDVisu landmarks | Other landmarks | Total |
| 32 (+) | 53 (-) | 85 |
| Different Developers | 61 (-) | 179 (+) | 240 |
| Total | 93 | 232 | 325 |

### B. Quantifying SDVisu’s Effectiveness

Following the centroid-based and sized-based heuristics, SDVisu identifies 284 landmark files from Mylyn’s code base. Compared to the 937 landmarks in the answer set (38.94% based on cycle analysis), SDVisu’s precision and recall is 60.9% and 18.4% respectively. This shows that SDVisu is fairly accurate in recognizing the navigational landmarks, but its result coverage needs improvement.

To gain concrete insights into SDVisu’s effectiveness, we specifically, consider those software change tasks
that have the contributions from both a single developer and multiple developers. For the same task, a landmark file is referenced by either the same developer or different developers. We therefore can use this analysis to assess the extent to which SDVisu extracts the concept-driven landmarks, i.e., the files referenced by different developers who work on the same task. In Mylyn’s dataset, 325 navigational landmarks based on cycle analysis satisfy our selection criteria, among which 93 are identified by SDVisu. We then perform a chi-squared test of independence [46] to determine whether there exists any association between developers and landmarks, since both variables are nominal. This inferential test is performed at \( \alpha = 0.05 \) level of statistical significance.

TABLE II displays the contingency table of the chi-squared test. The results are statistically significant (\( df = 1, \chi^2 = 4.6, p = 0.032 \)), which indicates significant associations exist. In Table II, positive and negative signs show positive and negative associations respectively. It can be inferred from Table II that the landmarks identified by SDVisu support the single developer in a positive way, whereas the landmarks identified by other means can be of positive influence on different developers working on the same task. In this sense, the concept-driven landmarks are best recognized by methods other than SDVisu.

C. Limitations

It is confirmed that revisiting is common and indispensable during code navigation. Although SDVisu is promising in identifying the landmarks emerged from developer navigations, other means need to be explored in order to increase the coverage of the identification. The construct validity of the Mylyn study is mitigated by the cycle-based operationalization of “developer’s navigational landmarks”. Similarly, the external validity is enhanced by considering a large-scale open-source software system and the code navigations carried out by the core developers in fulfilling the actual software maintenance tasks. Due to the statistical inference performed, internal validity [44] must be taken into account when interpreting the results. In terms of the results reported in Table II, internal validity refers to whether the differences observed are caused by the landmark-identification heuristics implemented in SDVisu. The most likely confounding variable is the existence of developer-centric landmarks, i.e., the files reflecting the individual code navigation idiosyncrasies. It is therefore not clear how much developer-centric landmarks contribute to the difference reported in Table II. Another major threat with our study design is related to our selection of the parameter values in operationalizing the navigational landmarks. Although the cycle length \( k \leq 3 \) and the window size \( n \leq 4 \) are adopted from the seminal work [2], adjusting the parameter values can affect the analysis results. Such adjustments may depend on factors such as the difficulty of the tasks, or the number of open editors as suggested by [2]. Finally, it is possible that the experience of developers has some influence on the use of the landmarks. For example, skilled developers are found to have longer navigation cycles [47], which may indicate more (developer-centric) landmarks are needed. It would be interesting to incorporate the developers’ difference into future analyses.

V. Discussion

As our results indicate developers’ reliance on the landmarks during code navigation, we discuss how our findings relate to other models for program comprehension and navigation. We then suggest design guidelines for incorporating landmarks and visual clustering in the tool support.

A. Implications for Theory

Our study elucidates the important role that legibility features play in facilitating developer’s spatial cognition and reasoning [35]. In particular, Cox et al. theorize a relationship between code navigation and real world spatial navigation by use of landmarks [7]. The spatial aspect of our work is rooted in Moonen’s work on software legibility [10] and relates to numerous efforts that model the code base as an information space, e.g., [48], [49]. Our findings also correspond to the earlier model on developers navigation. In Mylyn study, the navigation patterns are focused and repeated. As far as software clustering is concerned, we not only contribute the heuristics to identify the legibility features, but also observe a nice property of visual clustering. By using the LinLog energy model [16] to optimize the layout of the software dependency graph, the resulting clusters are mostly within the “bounded cardinality” [13] which ensures a reasonable cluster size to ease program comprehension. In [13], each cluster is set to contain between 5 and 20 software entities. As shown in Table I, the average size of Mylyn cluster is 16.2 and 27.3 respectively.

We believe the landmark reference model resulted from our investigation corresponds closely to the landmark-mapping model recently presented in [50]. In
Fig. 3. Design sketches for code navigation tool support (a): Integrating visual clustering to an SDE; (b) Zooming-in support for landmark-mapping.

observing end-users’ code searching of functional features, Gross and Kelleher realize that it is the landmarks between the code and the task that developers attempt to map [50]. The landmarks in Gross and Kelleher’s work refer to verbally identified points of interest. Our work improves the landmark identification via automated methods. Nonetheless, Gross and Kelleher illuminate the importance of recognizing the landmarks in the tasks. In their work, the task landmarks are elicited from observable program behaviors recorded in short video clips [50]. While program’s output is straightforward, our findings of concept-driven and developer-centric landmarks can lead to enriched ways for uncovering the prominent features in the task descriptions.

B. Implications for Tool Support

Although tools are only part of the complex nature of software engineering work, it is worth discussing how our findings can inform the tool development. We present here three design guidelines that help to integrate visual clustering into the SDE. Fig. 3 shows our design sketches.

- Interactivity. The tool shall enable developer’s interaction with it, e.g., zooming in to a more focused portion of the code base (cf. Fig. 3b).
- Multi-faceted path exploration. While SDVisu shows the usefulness of static dependencies, other means are sought by developers in code navigation, especially in determining entrance points [43]. Thus, the tool support should offer multi-faceted exploration paths, e.g., by leveraging structural (file hierarchy), textual (keyword search), runtime (debugger) information (cf. Fig. 3a).
- Explicit landmark-mapping. For tasks like feature location, corresponding the code landmarks with the task description’s prominent features is important [50]. In Fig. 3b, we also incorporate a selection area introduced in our recent traceability tool [37] in order to support the landmark mappings explicitly.

VI. CONCLUSION

Recovering software legibility features is an important reverse engineering activity. In this paper, we have explored the use of static-dependency-based visual clustering to uncover landmark files which can serve as reference points for developers’ code navigation. Our work not only advances the understanding about extracting legibility features of the software space, but also illuminates principled ways to increase the tool support. In the future, we plan to continue investigating the practical value of visual clustering in reverse engineering and program comprehension. Our immediate next step is to conduct qualitative analysis to gain further insights into developer’s code navigation behavior. Also, we plan to provide contextual information, e.g., by automatically generating cluster labels [51] [53].

ACKNOWLEDGMENT

We express our sincere gratitude to our collaborators for their valuable suggestions throughout the case study. Authors also would like to thank the anonymous reviewers for their constructive comments.
REFERENCES


