Visualizing the Evolution of Working Sets

Roberto Minelli, Andrea Mocci, Michele Lanza
REVEAL @ Faculty of Informatics — Università della Svizzera Italiana (USI), Switzerland

***Abstract*—As part as their daily work, developers interact with Integrated Development Environments (IDE), generating thousands of events. Together with other aspects of development, this data also captures the *modus operandi* of the developer, including all the program entities she interacted with during a development session. This "working set" (or context) is leveraged by developers to create and maintain their mental model of the software system at hand. Understanding how developers navigate and interact with source code during a development session is an open question.**

We present a novel visual approach to understand how working sets evolve during a development session. The visualization incrementally depicts all the program entities involved in a development session, the intensity of the developer activity on them, and the navigation paths that occurred between them. We visualized more than a thousand development sessions, and categorized them according to their visual properties.

I. Introduction

Integrated Development Environments (IDEs) are the main vehicles adopted by developers to build and maintain software systems. While the interaction with the IDEs apparently has the sole, ultimate effect of creating and modifying source code, this interaction also generates myriads of events that capture the various mechanisms of actual development. Not only is this *interaction data* not leveraged and made actionable in practice, but it is usually not even recorded.

There is increasing evidence about the valuable insights that can be inferred by mining interaction data, for example to understand how much time is spent by developers in performing activities like editing and program understanding, in IDEs like the PHARO IDE[1] [1] or VISUAL STUDIO [2]. In a more general setting, this data reflects the *modus operandi* of a developer, including the UIs and IDE components she uses more frequently [3], or enumerates all the program entities she interacted with during a development session.

According to Wexelblat [4], the information path obtained from navigating in an information space exposes and reveals the mental model of the system as perceived by a given user. In the case of software development, the set of entities navigated and interacted with, compose the "working set" (sometimes also called *context*) that developers leverage to create and maintain their mental model of the software system at hand supporting their current development task, *e.g.,* [5], [6].

Maintaining the working set is an essential part of program comprehension that absorbs a considerable portion of development time [7]. However, this process is often inefficient and not properly supported by IDEs. Many studies have shown evidence of issues related to navigation and the maintenance of working sets. A study by Fritz *et al.* [8], based on observations on three tasks solved by 12 different developers, found that on average the context model necessary to solve a task is composed of 4 classes, together with a subset of their methods. The study by Sillito *et al.* involved a variety of questions asked during maintenance tasks; these tasks included inspecting several entities, increasing the size of the working set [9]. A study conducted by Ko *et al.* found that 27% of the navigations concern visits to program entities that have been already visited, and that developers spend around 35% of their development time navigating source code entities [10]. The study also observed interesting patterns of *back-and-forth navigation* to compare related pieces of code. In our previous work we also found evidence of issues in navigation, as a likely manifestation of the problem of maintaining working sets [1]. More recently, we modeled and empirically measured the actual navigation efficiency of developers compared to different ideal scenarios, finding that there is significant room for improvement [11].

Some techniques have been proposed to improve construction and management of working sets. MYLYN, for example, leverages interaction data to build a degree-of-interest model (DOI), filtering the views of the ECLIPSE IDE from entities with a low DOI value [12], [13]. Other tools, such as NAVTRACKS [14], and TEAMTRACKS [15], monitor interactions to help navigation of software. NAVTRACKS supports this task by also visualizing related program entities with a simple graph.

We present a novel visualization to characterize how working sets evolve during a development session. The visualization is based on a force-based layout of a graph representing the methods and classes on the working set. The visualization depicts the intensity of the developer activity on entities of the working sets, and the navigation paths that occur between them. The visualization leverages data recorded with DFLOW, our interaction data profiler [1]. We visualized more than a thousand development sessions coming from 14 developers, and identified visual patterns on the evolution of working sets during development.

Our contributions can be summarized as follows:

- A novel visualization of the evolution of working sets;
- A visual analysis of over 1'000 development sessions, leading to a catalogue of visual patterns emerged from the analysis.

Structure of the Paper. Section II introduces the concept of *working set* and describes our visualization. Section III details a catalogue of patterns emerged from a visual analyses of development sessions. Section IV describes related work. Section V concludes the paper and outlines future work.

[1] See www.pharo.org

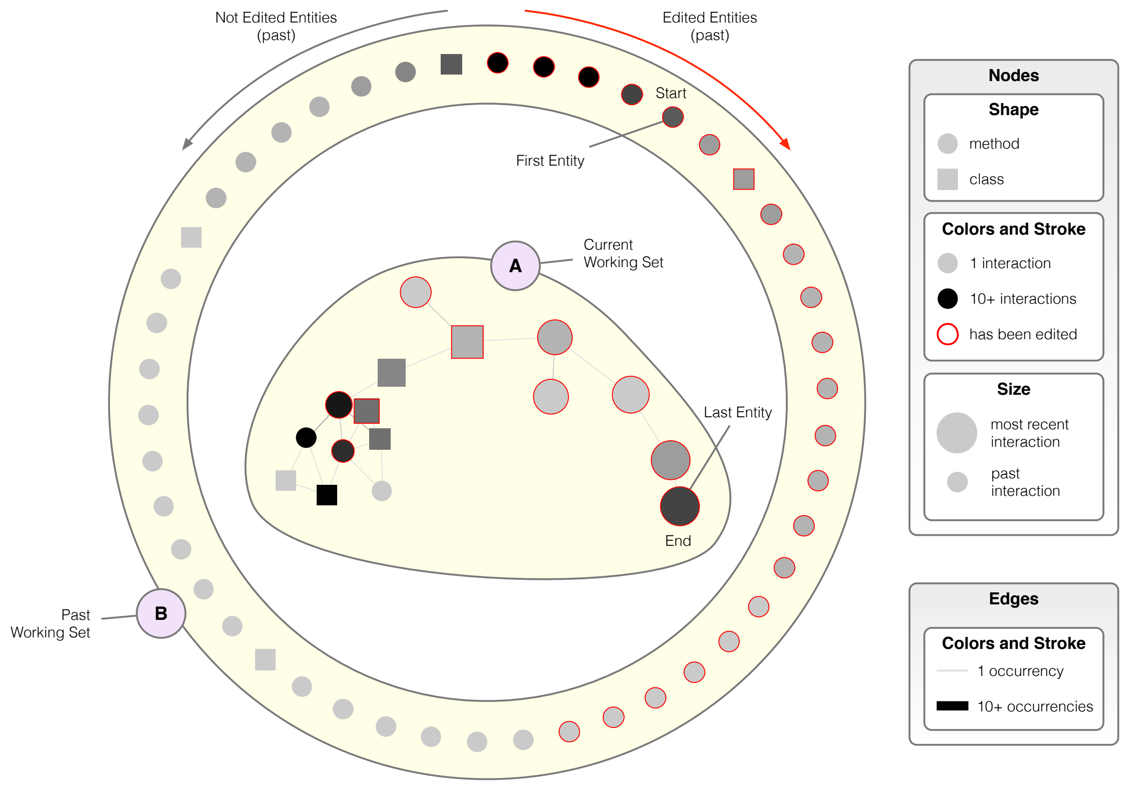

Fig. 1. Visualizing Working Set: Visualization Principles

II. Visualizing the Evolution of Working Sets

To better understand how working sets evolve during development sessions, we devised a novel visualization presented in Section II-B. First, we define the concept of *"working set"* (Section II-A).

A. *What is a Working Set?*

A *"working set"* is a group of program entities which a developer has interacted with during a particular period of time. We consider methods and classes as possible entities, and we consider the following interactions happening in the IDE, as recorded by DFlow: i) *navigation events*, *e.g.,* opening a class definition; ii) *edit events*, *e.g.,* modifying a method's source code; iii) *inspection events*, *e.g.,* checking the state of an object at runtime; we consider its class as the interacted entity. Our working set definition is similar to the *"task context"* defined by Kersten and Murphy [16].

We distinguish two kinds of working set:

- **Current Working Set.** The group of entities the developer interacted with during the last timeframe. The "last timeframe" can be defined in terms of number of interactions (*i.e.,* the last 30 interactions) or in a temporal fashion (*i.e.,* the interactions that took place in the last 10 minutes).
- **Past Working Set.** All the entities the developer interacted with in the past, before the *current working set.*

Our visualization depicts both working sets and their evolution during the recorded development session.

B. *Visualization Principles*

Figure 1 shows a development session depicted using our visualization. The view is made of two parts, depicting the current (Fig 1.A) and the past (Fig 1.B) working set respectively. The visualization is composed of nodes and edges.

Nodes. In the visualization, nodes represent program entities (*i.e.,* methods and classes) the developer interacted with during in a development session. Methods are depicted using circles, while classes are depicted using squares. Each node is colored using a gray-scale denoting the intensity of the interaction on the program entity it represents. A light gray node is a node with one (or a few) interactions. A node depicted in black is a node with 10 or more interactions, *i.e.,* the color scale saturates at 10 to make the visualization more simple to understand.

We distinguish two kinds of interactions: Interactions that do not modify the source code of the entity (*e.g.,* reading a class definition) and interactions that modify it (*e.g.,* editing the body of a method). The visualization adds a red border to the nodes that have been involved in at least one edit operation.

The size of each node depicts the recency of the interaction on the corresponding entity. By default, all nodes have a standard size of 20 pixels. The last node the developer interacted with has double the standard node size (*i.e.,* 40 pixels), and the nodes targeted by the last 10 interactions follow a linear scale from this double size to the standard size.

The visualization uses the labels *Start* and *End* to denote respectively the first and the last program entity the developer interacted with in the visualized interaction histories.

Edges. Edges express the flow of the interactions, and not structural properties of source code. For example, if there is an edge between `method Foo` and `class Baz` it means that, in the interaction histories, two subsequent events involved these two program entities (*e.g.,* a navigation event from `Foo` to `Baz`). Edges are undirected, *i.e.,* the edge between `method Foo` and `class Baz` summarizes all the flow of interactions between these two nodes, with no distinction whether the developer interacted before with `method Foo` or with `class Baz`.

Both the color and the stroke width of edges are mapped to the same metric, *i.e.,* the number of times the path represented by the current edge is followed by the developer in the interaction history. Both features are bounded. An edge depicted in light grey represents a path that is traversed one (or a few) times in the interaction history. A path depicted in black represent a path that has been crossed 10 or more times by the developer. The stroke width is bounded between 1 and 20 pixels (*i.e.,* that corresponds to the standard size of nodes). It follows a linear scale between the minimum number of occurrences of the path (*i.e.,* 1) and the maximum occurrences, computed using all the interactions of the entire session.

Layout. The view uses two layouts to depict current and past working set. For the current working set (Figure 1.A) we use a force-directed graph layout. Figure 2 depicts the underlying mechanics of the force based layout.

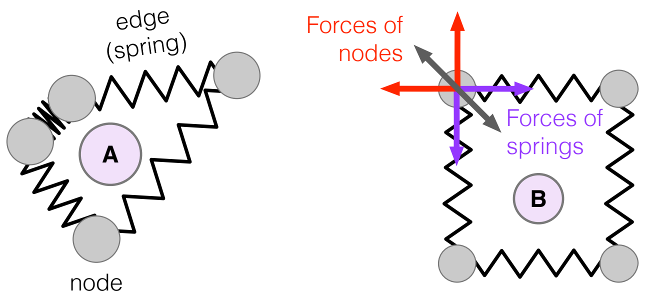

Fig. 2. Force-Based Layout Principles

Each node has a *charge* that tends to repulse other nodes. At the same time, edges act as *springs*, thus tend to assume their "ideal" length, *i.e.,* neither too compressed nor too spread. From an initial random configuration, depicted in Figure 2.A, the layout will progressively move the nodes trying to minimize all the forces exercised on the graph: the repulsive forces of nodes (*i.e., charge*) and the attractive forces of edges. Figure 2.B shows a configuration where forces are minimal.

At the beginning all nodes have the same *charge* and all edges have the same *strength* and *ideal length*[2], however as further discussed in Section II-C, these parameters evolve together with the evolution of the working set.

The outer part of the view (Figure 1.B) depicts the past working set using a radial (or circular) layout where nodes are equidistant. The radius of the layout depends on the inner part of the visualization, *i.e.,* the bigger is the space occupied by the inner part, the larger will be the radius of the circular layout for the past working set. Nodes are sorted according to the intensity of the interactions. Starting from the top of the circle (*i.e.,* 90°), nodes representing edited entities are placed clockwise, while the ones representing non-edited ones are placed counterclockwise.

Customization. All the fixed values described in this section are customizable. For example, one may easily change the number of interactions considered recent (*i.e.,* impacting the size of nodes) or the condition to distinguish between past and present working set.

Interaction. The view is highly interactive. The user can pan and zoom the visualization. By hovering on a node, the view shows a tooltip with additional information such as the name of the entity, the number of interactions and, if applicable, the number of edits on the hovered entity.

C. Co-Evolution of Working Set and Visualization

The final goal of our visualization is not present a static visualization but rather an animation of how the past and current working sets evolve from the beginning to the end of a development session.

To make this possible, we co-evolve the visualization with the working set, as explained in Algorithm 1.

```
Data: A sequence of events (i.e., InteractionHistory)
1  view ⟵ initialize an empty view
2  lastEntity ⟵ null
3  for event ∈ InteractionHistory do
4      currentEntity ⟵ extract the entity from the event
       // Adding or updating the node
5      if view contains currentEntity then
6          UpdateNode(currentEntity)
7      else
8          add currentEntity to the view
9      end
       // Adding or updating the edge
10     edge ⟵ edge from lastEntity to currentEntity
11     if view contains edge then
12         UpdateEdge(edge)
13     else
14         add edge to the view
15     end
16     ApplyLayout(view)
17     ApplyAging(view)
18     lastEntity ⟵ currentEntity
19 end
```

Algorithm 1: Constructing the View

First, for every event in the interaction history, the algorithm checks whether the program entity is already visualized and updates it, otherwise it adds it to the view.

[2]Node charge: -1,000; Edge strength: 1; Edge ideal length: 35

Second, it applies the layout and the aging mechanism on all the nodes and edges of the visualization, as described in Algorithm 2 and 3 respectively.

```
method UpdateNode (node) :
    update color of node (using # interactions)
    reset size of node (to max size, since last visited)
    reset time-to-live (TTL) of node (to default value)
    if node has been edited then
        add a red stroke
    end
    $charge \longleftarrow 80\% * charge$
end
```

Algorithm 2: Updating a Node in the View

Every time a node is re-visited, its color is updated with a linear grey-scale representing how many interactions have involved that entity. Its size is restored to the maximum size, since it is the last visited node. The algorithm also resets the time-to-live (TTL) of the *node* to the default value, *i.e.,* 30. The TTL is used to distinguish between current and past working set: When the TTL reaches 0, the *node* does no longer belong to the current working set. The last line of Algorithm 2 updates the node *charge*, used by the force-directed graph layout to arrange the current working set. At the beginning each node has the same initial negative charge; for how the layout is implemented, negative charges tend nodes to repulse themselves. For each interaction with an entity, we decrement its node charge by 20%, making the node less repulsive, and obtaining a more compact view of the current working set.

Algorithm 3 explains how we update each edge.

```
method UpdateEdge (edge) :
    update color of edge (using # interactions)
    update size of edge (using # interactions)
    $strength \longleftarrow 120\% * strength$
end
```

Algorithm 3: Updating an Edge in the View

An edge color is mapped to a linear grey-scale that represents how many times the edge has been walked by the developer. The same information is also encoded in its stroke width. The last step is the update of the edge *strength*, used by the force-directed graph layout for the current working set. The strength represents the force of attraction for the edges. A high value results in having nodes together. At the beginning each edge has the same initial strength (*i.e.,* 1). Every time the developer walks an edge, we increment its strength by 20%, bringing the two connected nodes closer.

Our approach considers the recency of the last interaction on a node as a key factor to determine the current and the past working set. Algorithm 4 explains this mechanism.

At each step we iterate over all nodes and we decrement their time-to-live (TTL). The size of each node corresponds to the recency of the last interaction on the node itself. Thus, at each step we decrement the size of each node (if its size

```
method ApplyAging (view) :
    for $node \in view$ do
        decrease the time-to-live (TTL) of node by 1
        if size of node $> minNodeSize$ then
            decrease size of node
        end
        // Disconnect node if TTL elapsed
        if TTL of node = 0 then
            disconnect the node from the graph
        end
    end
end
```

Algorithm 4: Applying the Aging Mechanism to the View

is not already below the *minNodeSize*, *i.e.,* 20 pixels). The last part of the method checks whether the TTL of a node is elapsed and disconnects it from the graph, *i.e.,* it removes all edges connected to it. In other words, if the node has not been targeted by any interaction in the last 30 iterations of Algorithm 1, it leaves the current working set.

Finally, the algorithm applies the force-directed graph layout for the current working set and radial layout for the past working set. Algorithm 5 illustrates this process.

```
method ApplyLayout (view) :
    $currentWS \longleftarrow$ connected nodes in the view
    $pastWS \longleftarrow$ disconnected nodes in the view
    $sortedPastWS \longleftarrow$ sort $pastWS$ according to
      number of interactions and edits;
    apply force-based layout to $currentWS$
    apply radial layout to $sortedPastWS$ (its radius is
      bigger than the $currentWS$, so that it fits inside)
end
```

Algorithm 5: Applying the Two Layouts to the View

After the aging process described in Algorithm 4, our approach can identify the current and the past working set. The current working set is composed of all the nodes that are connected in the visualization. To these nodes, we apply a force-directed graph layout, using the up-to-date *charges* and *strengths* computed in Algorithms 2 and 3 respectively. Among the advantages of this layout, the obtained visualization is aesthetically pleasing, simple, and intuitive.

The remaining, disconnected nodes represent the past working set. We layout the past working set with an equidistant radial (or circular) layout. In this layout nodes are evenly spaced between themselves to make it more aesthetically pleasing and intuitive. Circular layout also ensures that all the nodes are treated neutrally [17]. However, in our layout we made an exception to the rule by sorting the nodes according to the number of interactions and the editing status, *i.e.,* whether the represented program entity has beed edited in the past or not. This enable a quicker assessment of which entities, in the past, have been interacted (or edited) the most.

III. Visual Analysis: Metrics and Patterns

As a proof of concept, we visualized the evolution of a large set of development sessions collected with DFlow. Our visual analysis revealed a number of patterns referring to a single snapshot (see Section III-B) and evolutionary patterns (see Section III-C).

A. Dataset and Metrics

We applied our visualization to 914 development sessions, collected with DFlow, coming from 14 developers. Table I summarizes the dataset used for this study.

TABLE I
Dataset: Totals and Values Aggregated Per Session

All				*Total*
# Sessions				914
# Developers				14
# Snapshots				72,631

Per Session	*Avg.*	Q_1	*Median*	Q_3
# Snapshots	79.21	18	35	87
Working Set (WS)	9.57	4.70	7.13	12.10
Past WS	2.98	0.00	0.33	3.23
Current WS	6.59	4.00	5.86	8.78
Connectedness (%)	21.59%	15.67%	20.94%	26.78%

Our visualization of working sets in development sessions is evolutionary and incremental. Thus, for every session, we identify a number of *snapshots* to build a step of the visualization. We define a snapshot as a moment in time in which a program entity is either visited for the first time, re-visited, or modified. In total we identified 72,631 snapshots in our entire dataset.

The lower part of Table I reports the snapshot data aggregated per session. On average, each session has 79.21 snapshots (with a median of 35). The working set (WS), on average, is composed of 9.57 entities (on average 2.98 entities form the past working set and 6.59 the current working set).

The last metric we report is the percentage of *connectedness*. Given an undirected graph with n nodes, the maximum number of edges ($edges_{max}$) is:

$$edges_{max} = \frac{n \cdot (n-1)}{2}$$

Considering this as an upper bound, we can measure the percentage of *connectedness* of a graph with a given number of edges ($|edges|$) as:

$$connectedness\ (\%) = \frac{|edges|}{edges_{max}}$$

We compute the connectedness of the current working set. The connectedness expresses the average probability of two entities to belong to at least 1 subsequent pair of events in the recency window defining the current working set. On average our current working sets graphs have a percentage of connectedness of 21.59% (and a median of 20.94%). The most connected working set, not shown in the table, has a percentage of connectedness of 50%.

B. Snapshot Patterns

We discovered a set of interesting patterns that emerge by visually inspecting single snapshots of a development session, *i.e.,* a particular state of the past and current working sets of sessions. Below we discuss 5 patterns that we found.

Past: To Edit or Not To Edit

In our analysis we identified a number of session snapshots in which the past working set has a remarkable size but it contains no edit events (*i.e.,* no node in the past working set has a red stroke). Figure 3 depicts an example with 52 non-edited entities in the past working set.

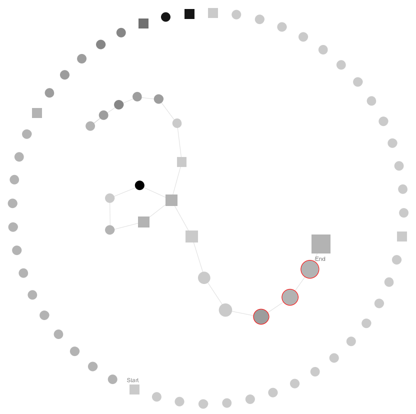

Fig. 3. Example of "Past: To Edit or Not To Edit"

This means that all the events performed in the past were explorative, targeted at the navigation of the system at hand. We conjecture that the developer needs to build her mental model prior to start her task, consistent with a significant time spent on program comprehension [7].

On the other hand, there are snapshots in which the past working set counts an high number of edits, as in Figure 4.

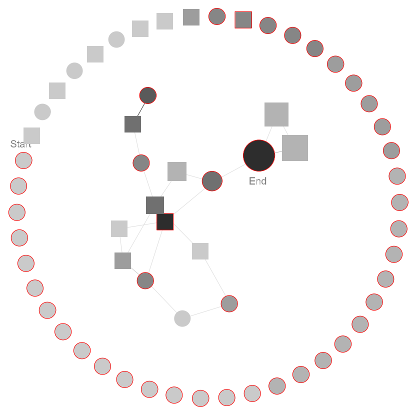

Fig. 4. Another Example of "Past: To Edit or Not To Edit"

In this snapshot more than 75% of the entities composing the past working set have been edited. Essentially, this could mean that the developer completed a given task and moved to a new one on separate entities.

U Can't Touch This

Among our sessions, there are snapshots which are entirely exploratory, *i.e.,* they lack edits both in the past and current working sets. Figure 5 shows an example of this pattern.

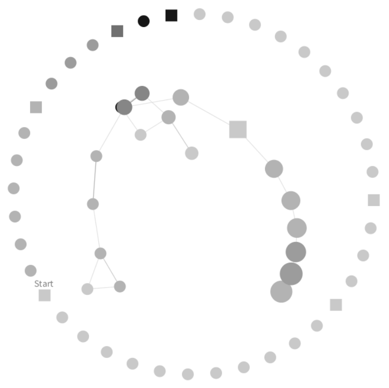

Fig. 5. Example of "U Can't Touch This"

Across the entire history, this session has, on average, a working set composed of 39 entities (22 in the past and 17 in the current). The figure depicts the 120^{th} snapshot (out of 147) of the session. Only in the last few snapshots the developer edits 3 program entities.

Potentially, the sessions manifesting this pattern are sessions in which the developer is addressing a complex task that requires a very deep understanding of the system, that is consistent, for example, with a complex debugging activity. After a deep phase of exploration, the developer has the necessary knowledge to perform few localized changes.

The Guiding Star

A development session potentially involves a large number of program entities. However, during development there might be a few landmarks that the developer uses as *guiding stars* for her exploration process. Figure 6 shows snapshot of a development session that clearly manifests this pattern.

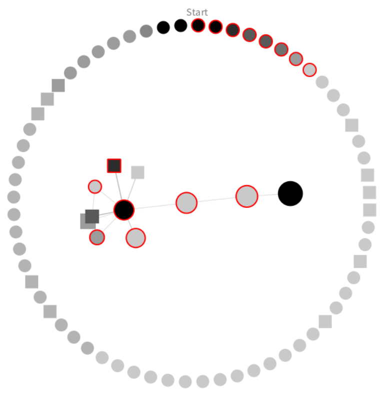

Fig. 6. Example of "The Guiding Star"

On the left side of the current working set we can see a circle depicting a method colored in black with 7 connected edges. This method likely plays a key role in the development session, or better in the current working set, supporting the exploration of 8 other entities, *i.e.,* 4 methods and 4 classes.

Another observation is that the number of entities colored in black is relatively low. Since the color represents the number of interactions, this is consistent with the fact the fact that developers need to periodically revisit some key entities (as observed by Ko *et al.* [10] and Soh *et al.* [18]), but also with the fact that the context model necessary to solve the task is often relatively small, *i.e.,* 4 classes (as observed by Fritz *et al.* [8]). Moreover, the edeges are relatively long, even the ones connected to the guiding star. This means that the cognitive jumps between the guiding star and a given connected entity are relatively few (since the edges are thin) and equally distributed among the related entities. This could be consistent, for example, with a small, limited refactoring.

Stay Focused, Stay Foolish!

Some sessions have a pattern similar to the guiding star, but involving a greater number of entities that are highly interacted with between themselves. In other words, the snapshot has a sort of "guiding constellation", where the current working set is highly focused on a set of entities, instead of a single one like the case of the guiding star. We call a working set focused if there is a subset of entities that are tightly connected between themselves and have a dark color, symptoms of a high number of interactions. Figure 7 shows a snapshot of a session manifesting this pattern.

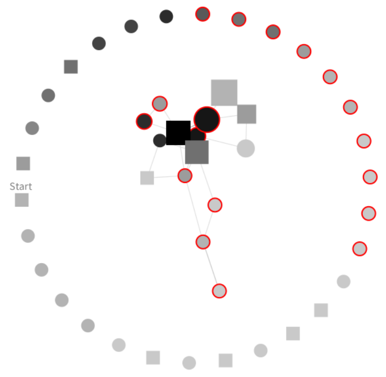

Fig. 7. Example of "Stay Focused, Stay Foolish!"

The top part of the current working set is very focused. Some nodes are very dark *i.e.,* they have been involved on a lot of interactions. Furthermore, they are tightly connected, meaning that there have been a lot of cognitive jumps between all the involved entities. Finally, we observe that the last interactions happen on a subset of the nodes in the focus (*i.e.,* some nodes are significantly bigger), meaning that this snapshot belongs to a task which is still revolving around the focused entities.

Moving in Circles

The last snapshot pattern that we present is called "Moving in Circles". As the name suggests, in the sessions manifesting this pattern, developers follow circular paths to explore and eventually modify the software system at hand. Figure 8 manifests this pattern.

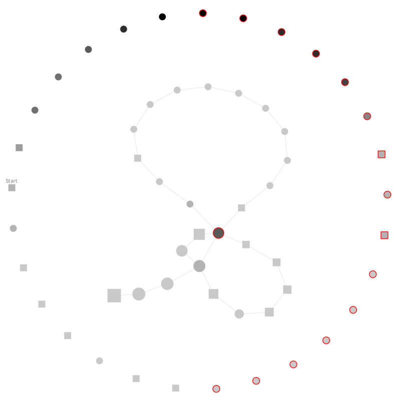

Fig. 8. Example of "Moving in Circles"

In the visualization above there are two large circular paths, one composed of 8 and the other of 14 entities. It is interesting to notice how all the entities composing the circular paths entity are only observed by the developer and never modified. The only modified entity in the current working set is the central dark grey entity that apparently acts as a small guiding star for the navigation. We conjecture that circular paths represent side exploration of the system at hand aimed at reinforcing the developer's mental model before—or during—the execution of a task.

A variation of this pattern sees edited entities inside circular paths, as exemplified in the session depicted in Figure 9.

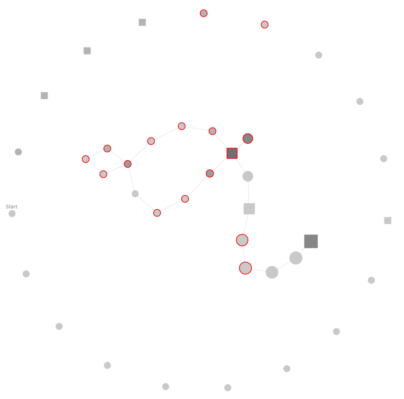

Fig. 9. Another Example of "Moving in Circles"

This can be consistent in a manual refactoring involving a sequence of methods of the same class, for example, that does not need to revisit the edited entities (*e.g.,*, in the case of a manual rename of a field).

C. Evolutionary Patterns

Evolutionary patterns consider multiple subsequent snapshots during the evolution of a development session. In this section we discuss 4 evolutionary patterns that we discovered in our visual analysis of development sessions.

The Past Awakens

During a session, the working set evolves: After taking part in the current working set, entities get old and move to the past working set. However we discovered that there are sessions in which entities also go through the reverse path: From the past (working set) they jump again into the current working set.

Figure 10 shows an example of "The Past Awakens". In *Part 1*, all the entities are in the current working set. Then, due to the aging process, in *Part 2*, the past working grows to 11 entities, accommodating all the entities that the developer is likely not to need in a short time. *Part 3*, instead, exhibits "The Past Awakens": From the 11 entities the past working set shrinks to 9 entities, symptoms that 2 entities have jumped back into the current working set.

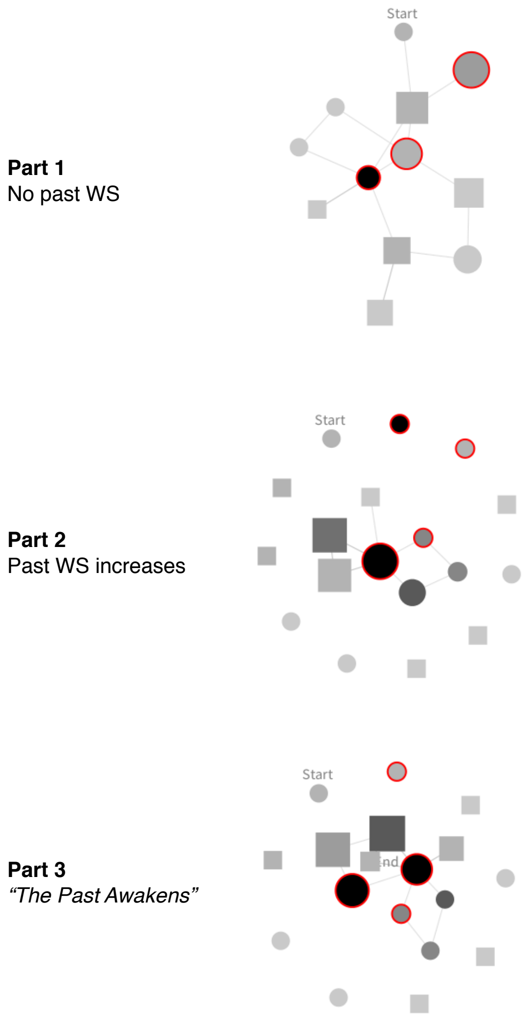

Fig. 10. Example of "The Past Awakens"

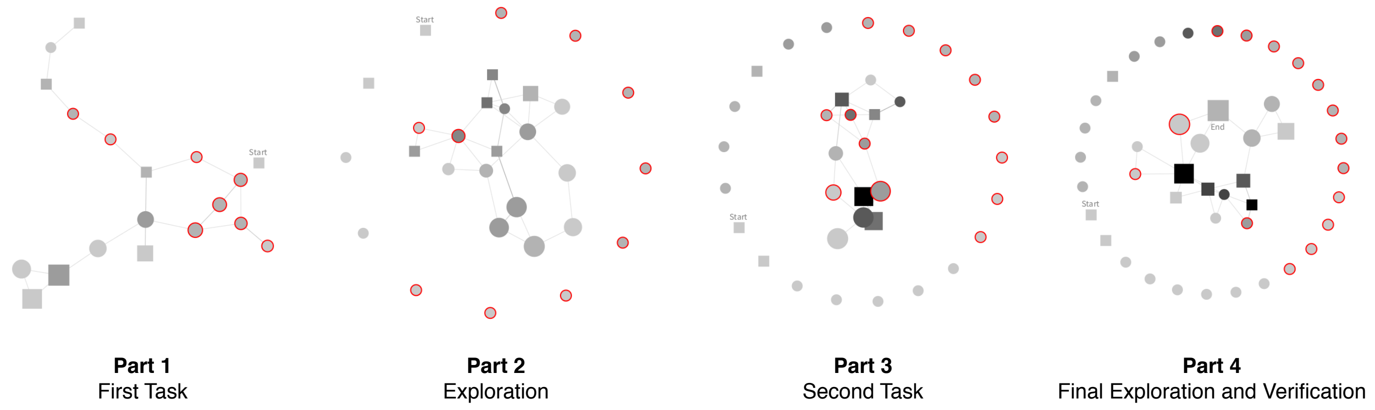

Fig. 11. Example of "Multi-Part Session"

The manifestation of this pattern adds evidence to the fact that, often times, developers need to revisit entities. According to Ko *et al.*, almost one third of the navigations target entities that have been already visited in the past [10]. By considering each of our snapshots as a form of "navigation" we can compare our data with their findings. On average, 4.53% of the snapshots of each session manifest this pattern. Even though this preliminary findings seems to contradict Ko *et al.*, the fact that an entity comes back from the past working set is more restrictive than a simple revisit.

Multi-Part Session

A development session is a sequence of conceptually related events happening in a relatively short timeframe. However, we can often identify clear subset of events that correspond to precise activities or phases. Examples include source code exploration, debugging, source code modification, etc. We call "Multi-Part Session" a session exhibiting this pattern.

Figure 11 shows an example of this pattern. In *Part 1* the developer addresses a task: She explores a set of entities and performs edit operations on 8 entities. In this first part, all nodes (except for the *Start* node that alone composes the past working set, *i.e.,* there are no edges connected to it) are in the current working set. In *Part 2*, the developer explores a different part of the system (*i.e.,* the past working set starts is not populated with 12 entities). She jumps from one entity to the other performing only 2 edit operations, possibly to augment or refine her mental model prior to performing a new task. In *Part 3* the developer edits two new entities and keeps interacting with some of the entities she has navigated in the second explorative part. The last part of the session (*i.e., Part 4* in Figure 11), is mostly explorative: All the edited entities go, or remain, in the past working set. Our conjecture is that in this last phase the developer explores the entities related to the ones that she modified during the session to verify the side effects of her modifications.

Our visualization supported us in visually identifying different development activities and interesting snapshots that otherwise would have been non trivial to find.

Thirst for Knowledge

Developer are often confronted with unfamiliar code or code that does not work and need to be fixed. When this happens they need to spend time in performing program comprehension and related activities. As depicted in Figure 12, this phenomenon is visible from our visualization that (mostly) portrays entities without the red stroke.

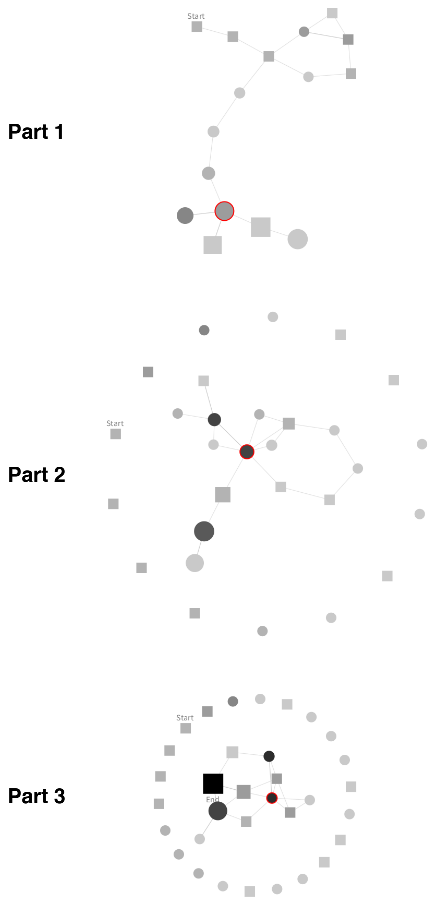

Fig. 12. Example of "Thirst for Knowledge"

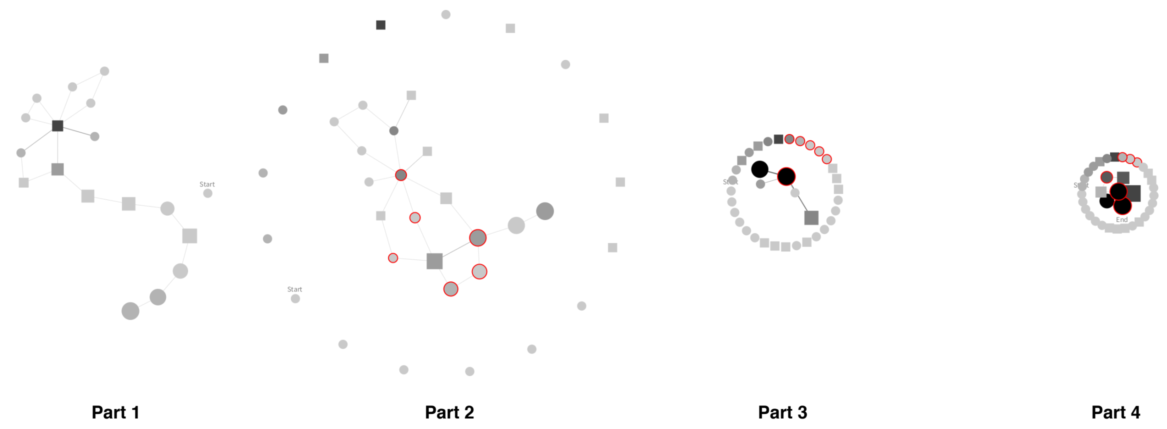

Fig. 13. Example of “The Working Funnel”

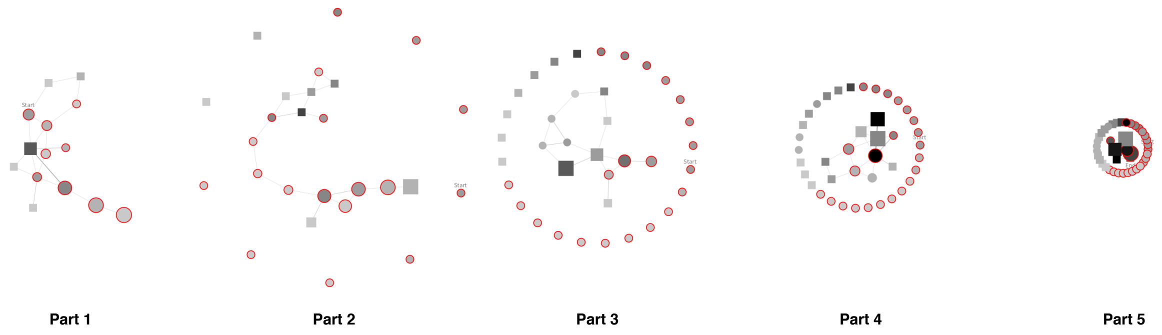

Fig. 14. Another Example of “The Working Funnel”

The Working Funnel

We observed that often the number of program entities that a developer interacts with in the initial phases of a development session is larger than the ones she interacts towards the end of the session. This can be attributed to several different factors. One possible reason for that is the fact that, prior to performing source code changes, developers need to gather a strong knowledge of the system by exploring it. As a result, in the initial parts of a session there are few edit operations but a lot of interactions, symptoms of an exploratory phase.

Figure 13 shows 4 snapshots of a development session that exhibits this behavior. In *Part 1* there are no edits, but a chain of explorative events. In *Part 2* the developer starts to modify a handful of entities, while continuing the exploration. In the remaining two parts of the session (*i.e., Parts 3* and *4*), instead, the number of entities in the current working set significantly shrinks. This is the symptom that the developer had stopped exploring. A possible explaination is that she is checking whether her modifications have the desired effects on the entities potentially affected by those changes.

A developer with a clear mental model of the system or that is facing an easy task, instead, could start by editing a large set of entities right away. Then, in a later part of the session, she could restrict her current working set to a handful of entities to perform the last, non trivial and more focused set of changes that finalizes her task.

Figure 14 depicts 5 snapshots of a development session that shows this scenario. In *Parts 1* and *2* the developer explores and modifies 21 entities. *Parts 3* is a steady phase in which the developer explores some entities and performs a few modifications. In the remainder of the session, *Parts 4* and *5*, the development flow calms down. In *Parts 4* there is still a bit of broad exploration (*i.e.,* the nodes in the graph are far away, symptom of a pure exploration phase). In *Parts 5*, instead, the working set is very narrow, nodes are mostly dark and very close between themselves. This means that the cognitive jumps are all focused on the current working set.

We call this pattern “The Working Funnel”[3]. In the sessions exhibiting this pattern, in fact, the working set is large at the beginning and progressively narrows down towards the end of the session, to guide the development flow.

D. Summing Up

We visualized the evolution of a large set of development sessions collected with our DFLOW tool [1]. The analysis revealed 9 patterns: 5 referring to a single snapshot of a development session and 4 evolutionary patterns.

In this section we showed how our intuitive visualization can support the identification of key entities in a development session, *e.g.,* “The Guiding Star”, or mechanism of the evolution of working sets, *e.g.,* “The past awakens”.

[3]A *funnel* is a pipe that is wide at the top and narrow at the bottom, used for guiding liquid or powder into a small opening

IV. Related Work

Researchers proposed different techniques to construct and manage of working sets. MYLYN, for example, exploits interaction data capture the task structure and building a degree-of-interest (DOI) model of program entities [19], [12], [13]. This model is then used in the ECLIPSE IDE to identify the entities that are more relevant for a task, *i.e.,* high DOI value.

NAVTRACKS [14], and TEAMTRACKS [15], instead are tools that monitor the interactions of the developer with the IDE to support navigation through software. NAVTRACKS also provides a simple graph based visualization to visualize how program entities are related.

Ying and Robillard used MYLYN data to characterize the editing behavior of developers with respect to the task they are carrying on [20]. They claim that IDEs can exploit this editing behavior to customize the views offered to the developer.

Soh *et al.* studies how developers explore software systems during maintenance tasks [18]. They characterized the type of exploration as either referenced or unreferenced. Among their findings, they discovered that developers mostly follow unreferenced exploration, *i.e.,* there is no set of entities that are visited with a higher frequency.

Fritz *et al.* conducted two observational studies performing change tasks to understand how big is the context model necessary to complete a change task [8]. Among their results, they discovered that code navigation models can differ substantially between different developers.

V. Conclusions

We presented a novel approach to visualize the working set of developers starting from interaction data coming from developer sessions and recorded with our profiler DFLOW. In particular, we leverage navigation, edit, and inspection events, and we provide an incremental, evolutionary visualization of the current and past working set. The visualization is based on the combination of two different and dedicated layouts: A radial layout for the past working set, and a force-directed layout for the current working set.

By visual inspection of the visualization, we identified several static and evolutionary patterns. In particular, we illustrated how our visualization can identify well-known like long, repeated navigations involving several entities, or the presence of central entities (that we called "guiding stars") that guide development tasks. On the evolutionary side, we identified patterns where entities on the past working set return to be subject of tasks later in the session, or cases where sessions are really composed of several independent tasks.

While reflecting on the characteristics of working sets is interesting to get insights about the mechanics of software development, we believe that our visualization can be more beneficial if integrated in the IDE and made actionable. Part of our future work, we envision a situation where the developer can leverage it to support alternative, possibly better navigation among program entities.

Acknowledgements

We acknowledge the support of the Swiss National Science Foundation (SNSF) for the project "HI-SEA" (No. 146734).

References

[1] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer – an investigation of how developers spend their time," in *Proceedings of ICPC (*23^{rd} *IEEE International Conference on Program Comprehension)*, 2015, pp. 25–35.

[2] S. Amann, S. Proksch, S. Nadi, and M. Mezini, "A study of visual studio usage in practice," in *Proceedings of SANER (*23^{rd} *IEEE International Conference on Software Analysis, Evolution, and Reengineering)*, 2016.

[3] R. Minelli, A. Mocci, R. Robbes, and M. Lanza, "Taming the ide with fine-grained interaction data," in *Proceedings of ICPC (*24^{th} *International Conference on Program Comprehension)*, 2016.

[4] A. Wexelblat and P. Maes, "Footprints: History-rich tools for information foraging," in *Proceedings of CHI (SIGCHI Conference on Human Factors in Computing Systems)*, 1999, pp. 270–277.

[5] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," in *Proceedings of ICSE (*28^{th} *International Conference on Software Engineering)*, 2006, pp. 492–501.

[6] M. A. D. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration," in *Proceedings of IWPC (*15^{th} *International Workshop on Program Comprehension)*, 2007, pp. 1–17.

[7] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[8] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich, "Developers' code context models for change tasks," in *Proceedings of FSE (*22^{nd} *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 7–18.

[9] J. Sillito, G. C. Murphy, and K. D. Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

[10] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[11] R. Minelli, A. Mocci, and M. Lanza, "Measuring navigation efficiency in the ide," in *Proceedings of IWESEP (*7^{th} *IEEE International Workshop on Empirical Software Engineering in Practice)*, 2016, pp. 1–6.

[12] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for IDEs," in *Proceedings of AOSD (*4^{th} *International Conference on Aspect-Oriented Software Development)*, 2005, pp. 159–168.

[13] ——, "Using task context to improve programmer productivity," in *Proceedings of FSE (*14^{th} *ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, 2006, pp. 1–11.

[14] J. Singer, R. Elves, and M.-A. Storey, "Navtracks: Supporting navigation in software maintenance," in *Proceedings of IWPC (*13^{th} *International Workshop on Program Comprehension)*, 2005, pp. 325–334.

[15] R. DeLine, M. Czerwinski, and G. G. Robertson, "Easing program comprehension by sharing navigation data," in *Proceedings of VL/HCC (IEEE Symposium on Visual Languages and Human-Centric Computing)*, 2005, pp. 241–248.

[16] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for IDEs," in *Proceedings of AOSD (*4^{th} *International Conference on Aspect-Oriented Software Development)*, 2005, pp. 159–168.

[17] F. Iragne, M. Nikolski, B. Mathieu, D. Auber, and D. Sherman, "Proviz: protein interaction visualization and exploration," *Bioinformatics*, vol. 21, no. 2, pp. 272–274, 2005.

[18] Z. Soh, F. Khomh, Y. G. Guhneuc, G. Antoniol, and B. Adams, "On the effect of program exploration on maintenance tasks," in *Proceedings of WCRE (*20^{th} *Working Conference on Reverse Engineering)*, 2013, pp. 391–400.

[19] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Čubranić, "The Emergent Structure of Development Tasks," in *Proceedings of ECOOP (*19^{th} *European conference on Object-Oriented Programming)*, 2005, pp. 33–48.

[20] A. T. T. Ying and M. P. Robillard, "The Influence of the Task on Programmer Behaviour," in *Proceedings of ICPC (*19^{th} *IEEE International Conference on Program Comprehension)*, 2011, pp. 31–40.