Interruption: Resuming Programming Tasks After Memory Failures and Interruptions

Christopher Ohara (406037)
ohara@campus.tu-berlin.de
February 28, 2019

Abstract—As software technologies become more sophisticated, more pressure is placed on programmers and software developers. In recent times, communication between stakeholders has drastically increased with the desire for regularly maintained software updates, custom functionality, and user-centered design aspects within a competitive environment. Errors and delays due to failures in cognition can be the end of a company. This paper primarily reviews two current approaches to helping programmers recover from faults related to (human) memory capabilities and the mitigation (prevention) of task interruptions). The general goal is to improve the overall productivity of an already overburdened programmer or developer. Parnin et al. suggest the usage of *smart memory aids*, designed to help the programmer target specific failures that are caused by particular cognition failures in the human brain. Abad et al. propose a model that is intended to identify and mitigate interruptions based on an empirical study. Ideally, if the causes of memory failure and interruptions can be correctly categorized, then it should be possible to improve the throughput of the programmer. The results of each paper are briefly described, a comparison is made, and considerations are given for future work.

Index Terms—Requirements Engineering, Memory Failures, Task Switching, Task Interruption, Productivity.

1 Introduction

1.1 Problem Statement & Motivation

Computer science and engineering are fields that very challenging and require years of priming and preparation. Programming, requirements engineering, and software development are fields that require a great deal of focus. Small errors can result in deadlines not being met or even worse, final products can be shipped with less functionality or security issues. Such issues can be the deciding factor on whether a company is successful or not, as these failures lead to unhappy users, safety issues, and not being the "first to market."

In modern times, programming teams are larger and interact with many departments and stakeholders. With the introduction of user-centered design, there is even more communication that influences the daily life and tasks of developers. However, human capabilities have not drastically improved to meet additional demands. Several aspects directly influence the error rate and time to completion for projects including memory failures, interruptions, and task-switching. While the human mind is sometimes viewed as a supercomputing device (human-computer analogy), it is prone to mistakes and failures. Furthermore, with the additional overhead from collaboration, programmers are finding their schedules fragmented leading to mainly short time periods (15-30 minutes) of programming and occasional sessions of one or two hours.

In "Programmer Information Needs and Memory Failures," Parnin et al. report that memory-related failures (primarily due to interruptions) lead to tasks requiring twice as much time and twice as many errors. Furthermore, one can consider that errors require additional time to repair[1]. This time spent prolongs finalization and also takes away from the programmer being able to complete other tasks (opportunity costs) that could lead to better functionality or implementation. Memory failures can arise from interruptions related to cognitive overheads. Abad et al. analyze commonly reported causes of interruption that plague requirements engineering. On both cursory and abstract levels, the issues that programmers and software developers (including requirements engineering) are effectively the same. Currently, researchers are attempting to resolve issues in resuming blocked tasks and memory failures effectively.

To counteract memory failures, Parnin et al. introduce specific *smart memory aids* meant to target errors that arise from different failures depending on the type of memory (roughly related to the function and area of the brain). Integrating several of these aids resulted in a prototype of a toolset (called *worklets*) though the reported results do not contain quantifiable data. Abad et al., on the other hand, report quantifiable data (based on qualitative surveys of developers) that give insights into their designs for a visualization framework model prototype. They distinguish the primary types of disruptions and their causes, as well as survey-based data to represent developer preferences in developing visual aids[2].

2 Background & Related Work

In this section, previous and related work information is briefly provided for context.

2.1 Memory Types

Essentially, memory can be categorized based on five different types related to the functionality and purpose. These types include prospective, attentive, associative, episodic,

and conceptual memory. Prospective memory is related to instances and reminders for conducting specific future tasks. Attentive memory is short-term has task-relevant items for a current goal or process (i.e. attention). As such, it is highly volatile (susceptible to interruptions and event-triggered disruptions) which is highly problematic for tasks that require focus on transient-related information. Associative memory is responsible for both creating associations and making real-time decisions regarding events that cannot be anticipated. These memories allow us to draw conclusions and expectations based on previous experiences. Episodic memory is primarily responsible for relying on past events (episodes) and creating long term representations of experiences. In some sense, it is similar to associative memory but with more emphasis on reflection and recollection. Lastly, there is conceptual memory, which is tasked with assigning objects and ideas to a higher abstraction level. Parnin et al. use the example of distinguishing the basic traits of a hammer (material, size, shape) to the concept of a ”tool.” The brain performs a type of labeling with feature engineering that results in abstract concepts. Since different memory types have different features, shelf-lives (volatility), and tasks, they will ultimately have different causes of failure (and approaches for remedy) as will be discussed below in the next entries.

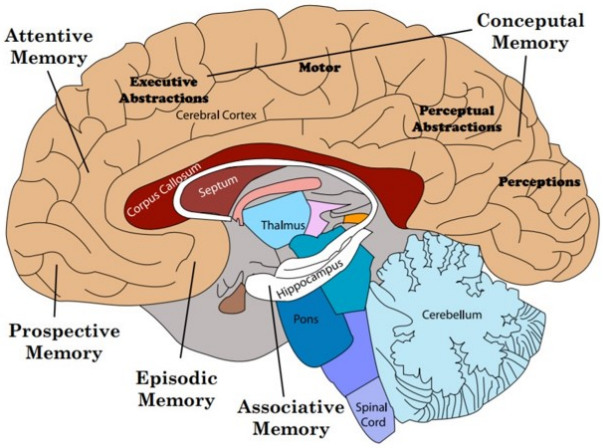

Fig. 1: Memory Types and their respective locations (sagittal perspective) [1].

2.2 Memory Locations

This section will briefly discuss the memory locations. The information provided is relevant for understanding how there is not a ”simple solution” to resolving memory failures. While memory types and locations can be loosely or tightly coupled (different areas of neurons can be activated in different regions for the same memory type), a simplified set of locations can be described regarding the primary activation areas within the brain. For instance, prospective memory is primarily located in the anterior prefrontal cortex. Attentive memories are activated in the ventrolateral and dorsolateral prefrontal cortex. The prefrontal cortex is considered the pinnacle of human ability evolution which allows for sentience and contemplation. Associative memory is embedded deeply in the brain (hippocampus), episodic memory is closely juxtaposed in the entorhinal cortex, and conceptual memories are primarily handled in two separate regions with a *perceptual* component in the posterior region and an *executive* component near the cerebral cortex.

2.3 Failures and Information Needs

Since memory has different functionalities and locations, it is intuitive to consider that cognition errors will result in different failures and information needs in order to recover. In this section, a singular entry (list) regarding failures and needs for each memory type will be provided for context. *Engage failures* are related to the inability to gain conscious attention (distractions, boredom). Task switching results in *concentration failures* which prevent the programmer from keeping focus on a particular task (interruptions). An *association failure*, as the name implies, is the failure to formulate associations with completeness (fragmentation). The inability to properly remember the specific sequences or details of an event or experience is called a *recollection failure* (long term memory specifics). Finally, *activation failures* are a result of not properly creating/calling concepts based on poor abstraction creation (interruptions, inaccurate priming).

Below, a short list is provided with the information need of the programmer (and respective failure):

- Prospective:
  - Programmers need facilities for modulating levels of engagement (Engage Failure)
- Attentive:
  - Developers need support for persistent focus (Concentration Failure)
- Associative:
  - Developers need support for building associations in code locations (Association Failure)
- Episodic:
  - Developers need support in recalling personal and social narratives (Recollection Failure)
- Conceptual:
  - Programmers need support in relevant concepts to promote priming” (Activation Failure)

As can be seen, appropriate countermeasures are needed to counteract (or prevent) failures. In the Solution section, a review of suggestions from Parnin et al. is provided regarding a prototype of *worklets* (based on *smart memory aids*). On the page below, a figure of a table is provided for all failures and needs.

2.4 Interruptions

An analysis of the types of memory failures demonstrates that there is a common culprit: interruptions. However, interruptions are not only a cause of memory failure, as they are also responsible for the overall performance and time required to complete a project (or task). Abad et al. refer to three primary definitions for the background of their work,

TABLE I: INFORMATION NEEDS AND MEMORY AIDS FOR DIFFERENT MEMORY FAILURES.

MEMORY	PROGRAMMING ACTIVITY	FAILURE	INFORMATION NEED	MEMORY AIDS
prospective	Resuming blocked tasks	Monitor failure Engage failure	Support monitoring applicability Provide multi. levels of engagement	*smart reminders*
attentive	Refactoring large code	Concentration failure Limit failure	Provide persisted and stateful focus Facilitate multiplicity	*touch points*
associative	Navigating unfamiliar code	Retention failure Association failure	Provide distinguishable features Support indexing by multi. modalities	*associative links*
episodic	Learning new API	Source failure Recollection failure	Store context Support narrative	*code narratives*
conceptual	Forming concepts	Activation failure Formation failure	Support priming Support abstraction	*memlets*

Fig. 2: Summary table for memories, activities, failures, needs, and aids [1].

so they will be introduced here. First, there is the concept of *disruptiveness*, which is the magnitude of the negative impact on a developers' productivity. As can be imagined, having a notification popup on a computer would be less disruptive than a fire breaking out in the work environment. Furthermore, interruptions have *interruption characteristics* related to the context of the task (type, progress, priority, timing, etc.). These characteristics can be given different weights. The third definition is related to *self-interruptions*, which were found to not only be highly common but overall very disruptive for requirements engineering (especially within task switching). An example of a state diagram is shown below in Fig. 3. In this figure, q_0 to q_f represents a task duration example when no interruptions are present. Taking the other 'path' results in an interruption (with a possible deadlock/trap) that requires additional time to complete. The *interrupting tasks* can have varying longevity and level of disruptiveness. Getting into the trap state (deadlock) shows that that the task is never resumed or never completed (essentially wasting more time).

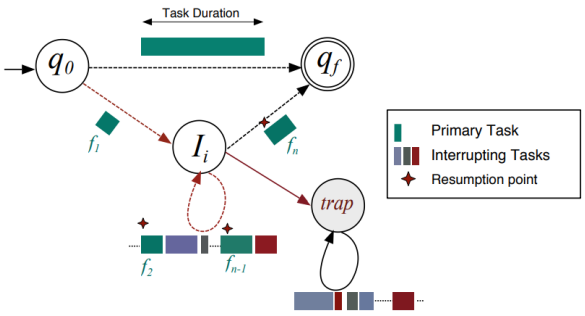

Fig. 3: A state diagram for a typical task execution [q_0 and q_f : initial and final states, f= task fragments, and I_i= the i th interruption]. The *disruptiveness* and *type* can hold different magnitudes (weights) as shown with the "Interrupting Tasks" chunks [2].

2.5 Visualization Techniques

Previously, research has been conducted on the effectiveness of visualization techniques (also by Abad et. al [4]) on differentiating various dimensions (that result in interruptions) from activities, stakeholders, and tasks. The goal is to identify the *focus* (defined as the *What, How, Why, Where* and *Who* components) of various visualization techniques as well as the *content* (specifically the activities themselves) (more information will be provided in the Solution section).

For *interruption visualizations*, Parnin et al. identified three main phases of interruptions: *suspension*, *resolution*, and *resumption* [3]. These phases have been adapted by Abad et al. to a more intuitive form: *before interruption*, *suspension period*, and *after interruption*. Essentially, this is the primary sequence during the interruption process and each section needs to be handled differently in order to prevent or mitigate interruptions (or at least the *disruptiveness* magnitude). Below, a figure from the previous work of Parnin et al. is shown to demonstrate the impacts of task switching and performance lag:

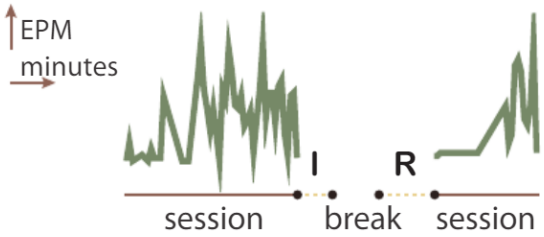

Fig. 4: Edits per minute (EPM), Interruption lag (I) and Resumption lag (R). During the brief period of interruption lag, the programmer may actively encode their mental working state to prevent an increased latency when resuming a task (resumption lag) [3]

3 SOLUTIONS

In this section, the two primary "solutions" proposed by the authors are described.

3.1 Worklets and Memory Aids

As we recall, different information needs will require different forms of memory aids. It is worth noting at this time that though Parnin et al. have proposed this solution (and prototype), they do not have concrete results for implementation (i.e., no quantitative data). Since resuming a block task (task switching) is mainly an issue for prospective memory, the suggestion is to use *smart reminders* (small popups that give a reminder at the appropriate time). Refactoring code requires concentration (attentive memory) and thus, a tool called *touch points* is introduced that is intended to maintain the status across multiple locations of code simultaneously (these are mostly similar to bookmarks). When navigating code that is unfamiliar, a framework called *associative links* can identify discrete features which assist in creating associations between different code locations and entries (based on modal properties, i.e., lexical combinations, syntax, etc.). *Code narratives* are an interesting concept that requires a deeper investigation, as Parnin et al. propose the ideas of *information quests* and *code replays*. *Information quests* are supposed to be a method of sharing the history of collected files (annotated with metadata) while *code replays* stream the events (narrative) of the programmer to others. It is

not clear how likely these would be adopted (or effective) in practice. Finally, the approach for conceptual memory is called *memlets* and this allows for programmers to map concerns within a workspace (like an inquisitive TODO). The combination of implementing these five memory aids is the goal of the toolkit *worklets*.

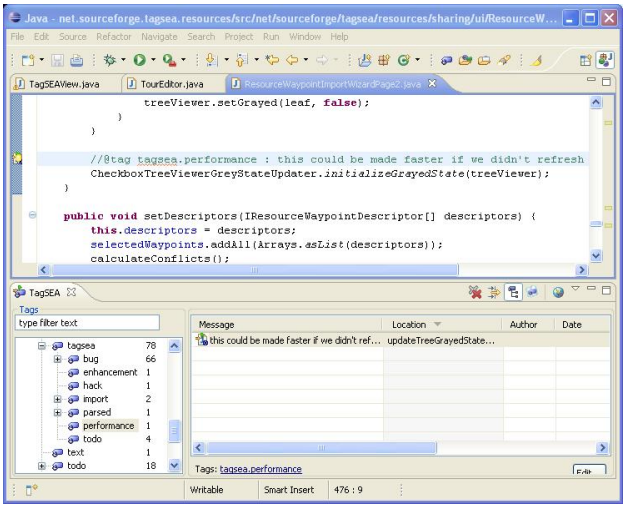

Fig. 5: An example of the software "TagSea" which creates intelligent tags that can be searched for in an IDE. These tags contain metadata and specifications about a particular module, snippet, or function. This is a framework related to the worklets prototype.

3.2 Interruption Visualization Framework

In order to create different visualization frameworks to handle interruptions, Abad et al. have considered implemented different *layers* based on the three aforementioned interruption timings. In their research, they collected surveyed data regarding the prevalence and type of interruptions that plague developers (which will be discussed in the Results section). The common theme is that different causes of issues require different approaches to a solution (i.e., there is no one framework that can effectively resolve multiple issues in either interruptions or memory failures).

The proposed framework breaks each interruption timing (before, during, and after) into a layer. Then, various goals, scopes (responsibilities) and artifacts are proposed for each layer. Prior to the interruption, various cues can be distinguished (at least, in retrospect) related to characteristics as broad as the time of the day (morning vs afternoon) to the amount of experience the developer(s) possess. For the *before layer*, this is considered in the scope of the overall "group" and requires some "narrative content-based visualizations." The underlying issue in this proposal is that the "narrative" is not well defined, except for that it is a "representation of the main transitions of the interruption graph." Therefore, it is difficult to evaluate or presume what such a narrative would look like.

During the interruption time period (the *suspension layer*), this becomes a shared responsibility between the *group* and *personal layers*, such that permanently suspended tasks need to be prevented. Using a notice-based visual component (popup notifications, sound effects). This improves the overall accountability of all that are responsible during a programming task (or even communication with other stakeholders and departments) as tasks are appropriately assigned and accounted for.

For the *after layer*, a time-centric visualization (annotation cues and thumbnail images) are triggered to assist the developer personally. This is intended to be a memory aid (though the framework focuses on task interruption prevention). Overall, the goal is to minimize the cognitive overhead related to interruptions on the *personal* level.

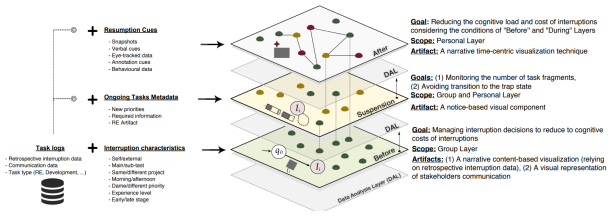

Fig. 6: The Proposed Interruption Visualization Framework by Abad et al. A larger version is displayed in the Appendix section [2].

4 Results

The primary focus of these works is to identify the causes and magnitude of memory failures and interruptions, as well as propose solutions, for programmers and developers. For the *worklets* proposed by Parnin et al., there are no quantifiable results (which is the primary reason that two papers were reviewed). The critical importance of their research is distinguishing the memory types (and failures) as this sets the framework for future solutions. Furthermore, they have proposed a prototype (though it was in 2012 and it does not seem to have been maintained), though better solutions can be found with IDE packages. The example of TagSea in Fig.5 demonstrates an unaesthetic and not very user-friendly design. Many IDEs (Atom, Sublime, etc.) have a much better implementation for tagging, bookmarking, reminders, and other items proposed in the prototype. The only issue is that there is no quantitative data related to how well these memory aids effect performance.

As for Abad et al., they did derive some quantifiable results. First, they gathered data via retrospective analysis for 5,079 recorded tasks based on 19 software developers. In term of interruptions, it was found that 66% of tasks had an extended delay (i.e., not resumed immediately). Furthermore, 11% of tasks were never resumed (i.e., a "trap" state or deadlock) leaving features abandoned or neglected. It was reported that developers need an average of 3.2 minutes after resuming a task to even recall what they were supposed to be doing (per task) demonstrating how detrimental task switching (or "multitasking") can be on the overall performance and throughput of a programmer or developer.

Another important result of their case studies was to identify the reported causes of interruptions. 30% of interruptions were caused by task-switching, usually as a

result of tasks being blocked due to a lack of information from other stakeholders (i.e., waiting on customer feedback, reviews, etc.). 26% of tasks were interrupted by the re-prioritization of other tasks (i.e., the previous discussed blocked tasks) due to a higher priority development. Astoundingly, 39% of tasks were caused by *self-interruptions* with 22% being due to "getting bored" and 17% from "personal schedules" (probably coffee breaks and meetings). The remaining 5% of tasks were classified as "other" with the quote of *"I usually get distracted by researching a new technology that could help solve the task at hand"*. If these causes categorized as "other" are also effectively *self-interruptions*, then approximately half of all interruptions are on the personal level.

In analyzing and surveying 53 software developers (with a mean experience of 4.5 years) for their proposed framework, they were able to identify user preferences (and weaknesses) in their model. As the narratives were not properly explained in the paper (as discussed previously), it would appear their implementation was not solid since 78% of participates did not like the approach (and the quote is *"no clue what I am looking at"*. Therefore, this approach is not very beneficial for programmers or developers in the near term future. Smart notifications were reported as the most preferred tool (81%) relying on popups, sound effects ("dings") or "encouraging" email reminders. However, users reported that these notifications should appear and ideal times. This is logical since an inappropriate or poorly timed reminder would either be ignored (ineffective) or distracting (counter-effective). Below, a full list of cues and visualization techniques is provided, since this review mostly covered the key aspects of the paper to prevent superfluous information recital:

(1= MOST USEFUL, 5= LEAST USEFUL)

Cues	1	2	3	4	5
Annotation cues	46%	17%	2%	21%	12%
Thumbnail images	15%	37%	26%	12%	10%
Verbal cues	24%	12%	17%	24%	22%
Eye cues	12%	20%	22%	24%	22%
Behavior graph	5%	15%	29%	17%	34%

Fig. 7: Participants' responses to the usefulness of various resumption cues [2].

tools can be integrated seamlessly into the daily activities of programmers and developers (i.e., probably not some form of narrative...).

As programmers are required to complete an increasing number of non-traditional requirements (i.e., interacting with the user or implementing techniques from data science, etc.) these issues will continue to increase. While many technical people identify as 'multi-taskers,' evidence shows that they are not very effective at it. As such, many causes of memory failures and interruptions have been identified. It would be highly beneficial for companies (and individuals) to find methods to prevent catastrophic forgetting and boredom in programmers.

Recently, researchers at the Karlsruhe Institute of Technology (KIT) have been conducting similar research related to graphical user interface usage by creating dashboards and utilizing eye-tracking sensors (Toreini et al) [2]. The goal is the same; track and optimize performances yet the focus is on individuals (less emphasis on groups). However, similar to the other research presented in this paper, they do not have quantitative results reported. Toreini et al instead report "a small pilot test study [had] preliminary results [that] showed an improvement in the task resumption performance by decreasing the resumption lag." An interesting aspect is to how much deviation and uncertainty must appear in such test beds, if researchers do not publish the results.

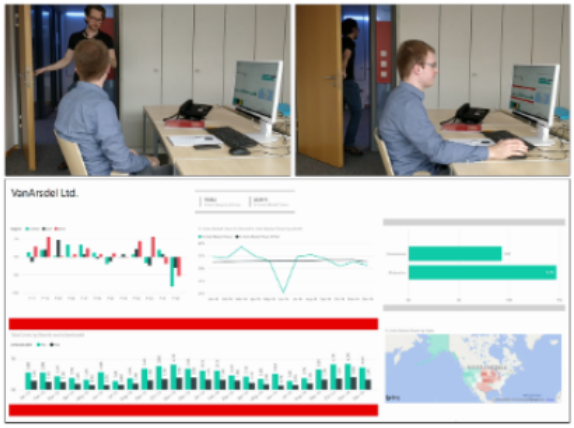

Fig. 8: The attentive information dashboard detects there visit of the user and provides two types of feedback after return to the primary task [5].

5 Conclusion

5.1 Summary

In summary, it can be seen that there are (currently) many issues regarding the limitations of human performance. Memory failures can lead to twice as many mistakes and require double the time to repair. Nearly half of all interruptions are caused by *self-interruptions*. Interruptions require an average of more than three minutes for the programmer to even recall what they were working on. Some of the solutions proposed for resolving either challenge might, in fact, lead to more errors and delays unless these frameworks and

5.2 Future Work

As alluded in the Results section, quantitative data regarding the effectiveness of the proposed frameworks and tools is not available. These tools attempt to mitigate the causes of errors and failures with discrete or specialized tools. However, without knowing how effective these tools are, one cannot assume that they will necessarily be a boon to performance. What is more, is occasionally packages and toolkits require time and habit-forming in order to be effective over the long term. This is usually not a safe practice to engage in whenever there are impending deadlines.

Therefore, users of these tools would likely need experience using them prior to a major career position in industry (as I have heard many senior software developers even state they still program in VIM or NANO since they do not like IDEs...).

Another consideration regarding future work is analyzing the actual effectiveness of using specialized packages and themes in modern IDEs. Particularly, there are now features that can use auto-complete, TODO tagging (with exporting to SCRUM for Agile development), error recognition, automatic "help" features, multiple line/character editing/shifting, and so on. Quantitative data can easily be acquired by creating an experiment with a controlled variable (of programmers) in which they are asked to program a project normally when using additional IDE features. Afterward, they would be asked to program for a time without such features. Given a large enough sample size, more objective data could be completed.

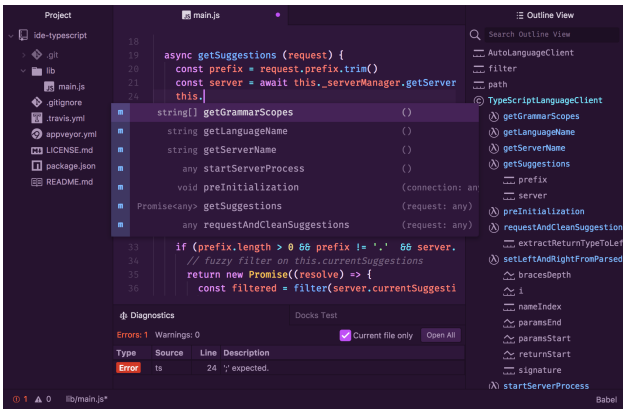

Fig. 9: Auto-complete and error identification in Atom IDE.

Overall, it is critical to remember the goal of removing interruptions and memory failures. Essentially, we want to improve the performance and reduce the cognitive load of programmers. Therefore, any framework, tool, or package needs to be integrated seamlessly into the work environment and habits of the programmer. As a result, performance and throughput will increase regardless of the causes of issues. It is still, of course, ideal to resolve these issues since they are prevalent and carry notable consequences.

6 References

[1] C. Parnin and S. Rugaber, "Programmer Information Needs After Memory Failure," in *Proceedings of the 2012 IEEE 20th International Conference on Program Comprehension (ICPC)*, Jun. 2012,
DOI: 10.1109/ICPC.2012.6240479

[2] Z.S.H. Abad, A. Shymka, J. Le, N. Hammad, and G. Ruhe, "A Visual Narrative Path from Switching to Resuming a Requirements Engineering Task," in *Proceedings of the 2017 IEEE 25th International Requirements Engineering Conference (RE)*, Sep. 2017,
DOI: 10.1109/RE.2017.81

[3] C. Parnin and S. Rugaber, "Resumption Strategies for Interrupted Programming Tasks," in *Software Quality Journal, vol. 19, no. 1, pp. 534*, 2011.

[4] Z. S. H. Abad, M. Noaeen, and G. Ruhe, "Requirements Engineering Visualization: A Systematic Literature Review," in *Proceedings of the 2016 IEEE 24th International Requirements Engineering Conference (RE16) pp. 615*, IEEE, 2016,

[5] P. Toreini, M. Langner, and A. Maedche, "Use of Attentive Information Dashboards to Support Task Resumption in Working Environments" n *Proceedings of the 2018 ACM Symposium on Eye Tracking Research Applications Conference*, Jun. 2018,
DOI: 10.1145/3204493.3208348

7 Appendix

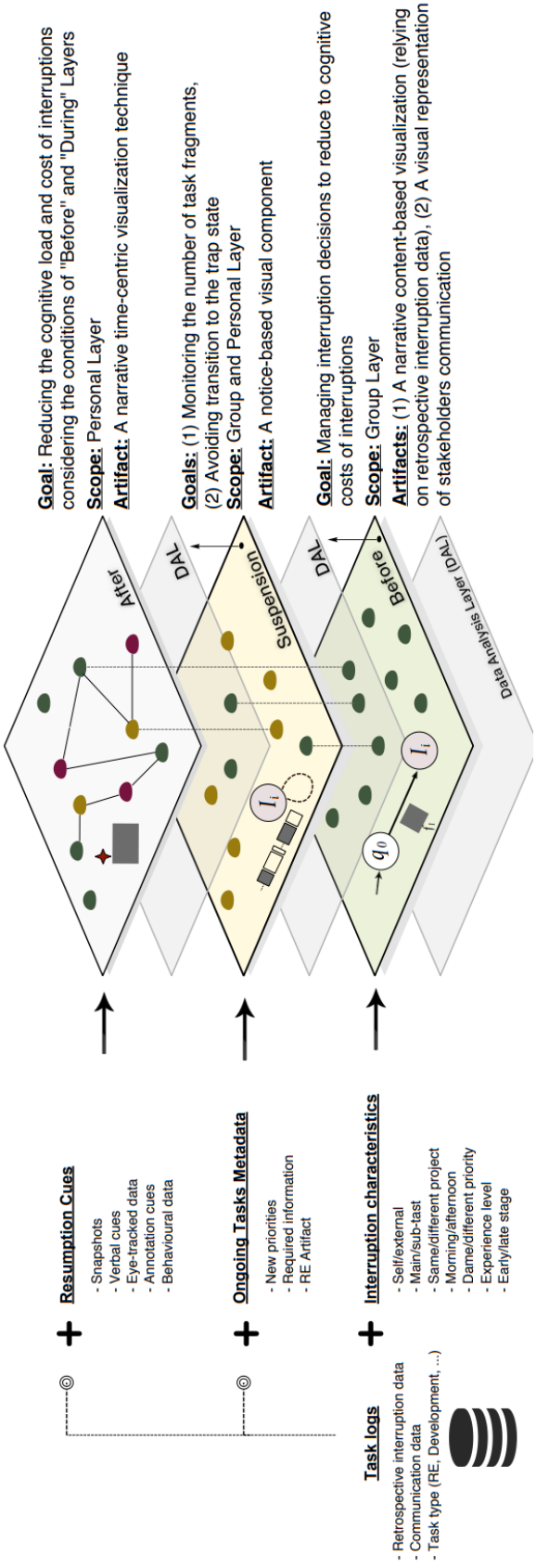

Fig. 10: The Proposed Interruption Visualization Framework by Abad et al. [2].