In P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds). Proc. PPIG 17 Pages 173 - 187

Representation-Oriented Software Development: A cognitive approach to software engineering

John J. Sung

School of Humanities, University of Sussex, Falmer, Brighton BN1 9QN, UK
j.j.sung@sussex.ac.uk

Software development is necessarily a cognitive process. Software engineers (cognitive entities) develop software to maximize productivity while delivering quality software on time. In essence, issues in software engineering can be conceptualized as a cognitive optimization problem. The utility of this approach is illustrated by an example in which a cognitive semantic approach is applied in analyzing the relationship between representations used in describing aspect-oriented programming (AOP) [14]. This approach is applied within the framework of distributed cognition such that humans and computers involved in the software development are conceptualized as one cognitive system that may be optimized. Optimization of the cognitive system involves analyzing the human cognition in understanding software and opportunistically offloading the identified human cognition onto the computer. The analysis of AOP leads to represented-oriented software development (ROSD), in which the problem of creating, manipulating and maintaining representations is its main concern.

Introduction

Software development is necessarily a cognitive activity. This fact is the motivation for applying methodology and concepts from cognitive science in designing user interfaces [1,2]. Currently, software engineering [3] is mostly a collection of heuristics and no underlying theory exists to analyze and optimize the software development process. Much of this characteristics of software engineering may be attributed to the fact that a theory of software development has to be based some theory of human cognition. To address this shortcoming of software engineering, this paper applies concepts such as distributed cognition [4], cognitive offloading and conceptual integration [5,6,7] to argue for a cognitive approach to software engineering. In order to understand how this cognitive semantic approach can solve problems in developing software, it is important to understand particular issues in software engineering, such as the problem addressed by an emerging software development paradigm called aspect oriented software development (AOSD).

Aspect Oriented Software Development

Object oriented technologies are utilized widely in modern software development. However, improvements are needed in software practices to handle ever-increasing demand on software. Aspect oriented software development (AOSD) [8,9] is an approach that attempts to address this issue from several observations made about software development. In many cases, developers have to trade-off between clean and easy to understand implementation with a clear functional decomposition and optimized implementation that are tangled and hard to understand. The code is tangled in a sense that the optimized code is the result of merging several functional components together, making the code hard to understand and change. Therefore, a way of addressing this issue is to create some base code that is easy to understand and modify. Then, directives are added to change the base code to optimized code or add functionality. Aspect oriented programming (AOP) approaches such as AspectJ [10] and composition filters [11] take this modification of base code approach. Demeter [12] and HyperJ [13] take another approach in which base code is not required. The program is written such that there is no dominant decomposition. In a sense, the linguistic features of Demeter and HyperJ provide directives in weaving the object-oriented code. In all of these AOSD approaches, the main focus is the separation of concerns, a different way of modularizing programs.

For the purpose of this paper, concerns are properties or artifacts of the software that may be desirable. The most prototypical concerns in AOP are properties such as logging, error handling, performance and persistence. Then, the separation of concerns becomes the separation of desired artifacts or properties of the software being developed. Essentially, the separation of concern is a vague notion that encompasses any type of modularization of the software for the purposes of adaptability, maintainability, extendibility, and reusability. This vague definition for separation of concern allows for more flexible modularization of programs than with OOP in which programs are modularized into class hierarchies. In the end, the goal of all AOSD approaches is to allow programmers to define and manage these concerns in such a way that software development process is improved. I will argue that this goal of separating concerns is part of the human cognition that allows understanding to occur. When this is taken within the framework of distributed cognition, the problem of designing tools for software development becomes the problem of organizing representations for optimizing the distributed cognitive system.

Optimizing Distributed Cognition

Hutchins [4] argues for a distributed view of cognition. The complexity of navigating a navel vessel requires distribution of cognition such that the crew of the ship may safely navigate a ship. The task of navigating a ship is distributed among the crew and every member of the crew has a particular task to perform. Specific protocols and artifacts facilitate communication for coordinating the distributed cognitive system to navigate the ship. Similar to ship navigation, cognition for developing software is also distributed. Depending on the software project the distributed system could be a single engineer and a computer or a team of engineers

and a network of computer systems. Therefore, a distributed cognition view of software engineering would construe the various people, computers and environment as part of the distributed cognitive system in developing software. For the purpose of this paper, the cognitive system is construed as a software developer and a computer, with representations playing a crucial role. The understanding of how representations are used in software development provides a method for analysis and optimization of this cognitive system.

The conceptualization of the cognitive system as the computer system and the developer shifts the priority of the software engineering to the optimization of the cognitive system by distributing cognition among the different parts of the cognitive system for software development. Any type of optimization requires that one analyze the processes for critical paths and find opportunities for minimizing computation and memory usage. There exist many methods for analyzing the computation and memory usage of computer programs. However, there are no methods for analyzing human cognition to optimize the cognitive system as defined above. Latest cognitive science research could provide the theoretical foundation for such an analytical method for cognition. In particular, conceptual integration theory (CIT) [5,6,7], a theory of meaning construction from cognitive linguistics [15,16,17], will provide a method for analyzing cognition during software development. CIT provides a method for describing and identifying human cognition that may be offloaded onto the computer. I am assuming that offloading cognition from the human to the computer in general will optimize the cognitive system, i.e. improve software development.

Conceptual Integration Theory

Conceptual integration theory (CIT) is a theory of meaning construction arising from cognitive linguistics. It describes how humans integrate various information from our sensori-motor system to create meaning. Generally, CIT is used to explain how we understand language, diagrams and pictures. The explanation takes the form of an integration network that describes what is necessary in order to understand the linguistic utterances, diagrams, or pictures. Therefore, it does not describe what actually happens, but characterize the sort of cognition that would produce the observed phenomenon. This is adequate for the purpose of analyzing human cognition to optimize the human-computer cognitive system.

Representations

Another aspect of CIT that is important for this paper is the fact that CIT may be used to analyze not only language, but also other stimuli that may be meaningful. In software development, there are many different types of representations used, representations such as class diagrams, sequence diagrams, collaborations, etc. The definition of representation for this paper is not limited to these program specific artifacts. Representations also include other linguistic artifacts such as documentation, specification, and requirements. The particular representations analyzed in this paper are program code and diagrams that are used in explaining the optimization of an

image filter in [14]. Cognitive semantic analysis of understanding the optimization of the image filter shows that representations are interrelated and this interrelation as described by the conceptual integration network is the result of human cognition. Therefore, these interrelations between representations are necessary for the software developer to understand the implementation and identify possible optimizations. The cognitive process of interrelating is the cognition that we are interested in offloading onto the computer. This would make it logical that a purpose of software engineering should be to identify, describe, modify and manage these representations used during software development to optimize the human-computer cognitive system by offloading the cognition required to understand the interrelation between these representations. The first step towards this approach in which representations are crucial is the short introduction to the conceptual integration theory below.

Conceptual Integration Theory

Conceptual integration theory is a theory of how we are able to create meaning from various stimuli. It uses concepts such as mental spaces, mental space elements and connections to characterize how meaning is constructed. This section introduces CIT through the "clipper ship" example from Fauconnier and Turner [7]. In 1993, a modern catamaran is attempting to break the record for sailing from San Francisco to Boston that was set in 1853 by a clipper ship. A sailing magazine, *Latitude*, describes the progress of the catamaran in this manner:

> "As we went to press, Rich Wilson and Bill Biewenga were barely maintaining a 4.5 day lead over the ghost of the clipper *Northern Light*, whose record run from San Francisco to Boston they're trying to beat. In 1853, the clipper made the passage in 76 days, 8 hours." - "Great America II," *Latitude 38*, volume 190, April 1993, page 100

According to the conceptual integration theory, one has to use concepts such as mental spaces and connections, in order to describe how one understands this particular passage from *Latitude*. *Mental spaces* are small conceptual packets that are constructed for local understanding and action. In this example, there are two input mental spaces (*input space*) that provide the elements for conceptual integration. The first input space contains the clipper ship that made the voyage from San Francisco to Boston in 1853. The second input space contains the modern catamaran making the same trip in 1993. Between the elements within the two input spaces, there are connections that relate them. These *connections* may be any type of relations that are relevant for the analysis, such as over, under, father of, Identity, Representation, etc. The relevant connection in this example is Analogy[1]. There are analogical cross-space connections between Boston of 1853 and 1993, the clipper ship and the catamaran, San Francisco of 1853 and 1993, and the voyage that the clipper ship and the catamaran took.

[1] Vital relations as described in [7] are capitalized, following Fauconnier and Turner's convention.

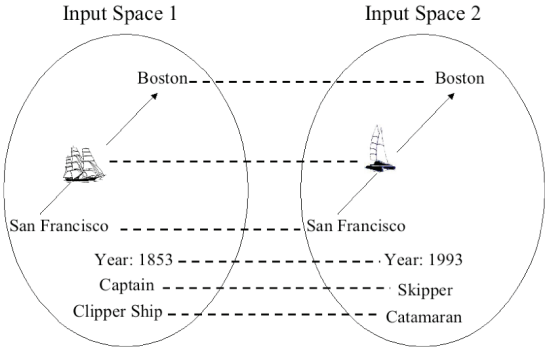

Fig. 1. Input Spaces and Connections for the "Clipper Ship" Example

There could be other elements within the mental spaces, such as the captain of the clipper ship or the skipper for the catamaran and an analogical connection made between these two elements. Therefore, the elements presented here are not exhaustive. The input spaces and the cross-space connections between elements only describe the how we are detecting the similarity between the voyage made by the clipper ship in 1853 and the voyage made by the catamaran in 1993. It does not describe how we understand the catamaran's voyage as a race against the "ghost" of the clipper ship. In order to describe this process, we have to use the concept of blended space. *Blended spaces* are mental spaces in which elements are projected and some operations take place. In this example, the cities Boston and San Francisco, the year 1993, the catamaran and the clipper ship are projected onto the blended space. Notice that many of the elements are fused (such as the city of Boston in 1853 and in 1993 are treated as one city even though they are very different), not projected (such as the year 1853), or ignored (such as the differing sea conditions between the two voyages). An important operation that occurs within the blended space is called elaboration. *Elaboration* is a mental space operation that allows for "running" of the blend. To understand the statement, "If the catamaran keeps its lead, the catamaran will win the race," we would have to run the blend, i.e. mentally simulate the boat race to Boston. The conclusion of the elaboration would result in the assertion that the catamaran will win.

According to the conceptual integration theory, construction of an integration network is meaning construction and the integration network consists of input spaces that provide elements for conceptual integration, cross-space connections between the elements in the inputs spaces, a blended space that contain projected elements from the input spaces, and operations are applied to the projected elements in the blended space. This integration network is able to explain how we are able to understand the race between a modern catamaran and a clipper ship even though the clipper ship took the journey a century earlier. The conceptual integration theory described through the "clipper ship" example has been a basic introduction and further details of the theory should be obtained from [5,6,7].

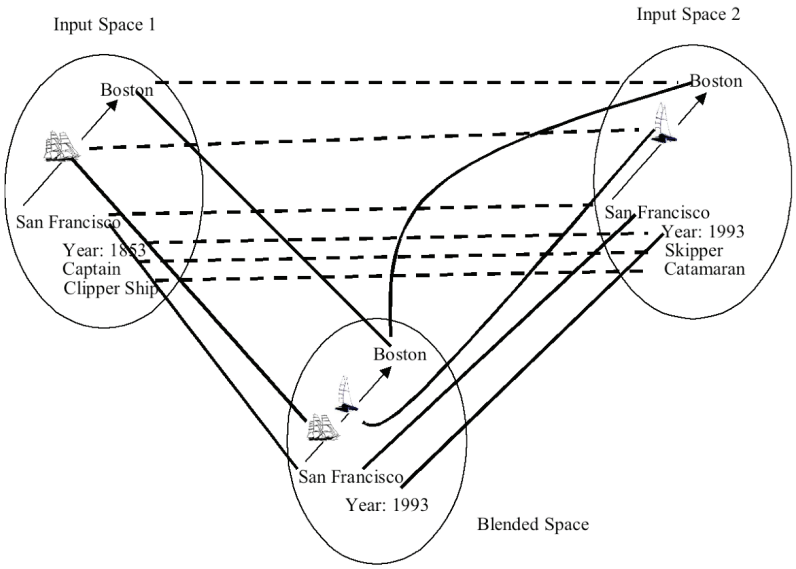

Fig. 2. Integration Network for the "Clipper Ship" Example

Optimization of an Image Filter

In this section, the conceptual integration theory introduced above is applied in analyzing the cognition required to understand the image filter optimization example in [14]. Kiczales et al. uses this image filter optimization example to show that there is a trade-off being made between easy to understand and inefficient code, and hard to understand and efficient code. They argue that an aspect oriented code would be both easy to understand and efficient. This is achieved by providing linguistic facility that describes the way in which the code should be optimized. They call these descriptions for transforming the base code, aspects. The motivation for aspects that allow separation of concerns is shown by the image filter optimization example.

The image filter shown in Fig. 3 is the `or!` filter that loops over each pixel within two images to apply the logical OR operator to the corresponding pixels. The implementation in Common LISP allocates a new image, then loops over each row and column of the image to perform the logical OR operator.

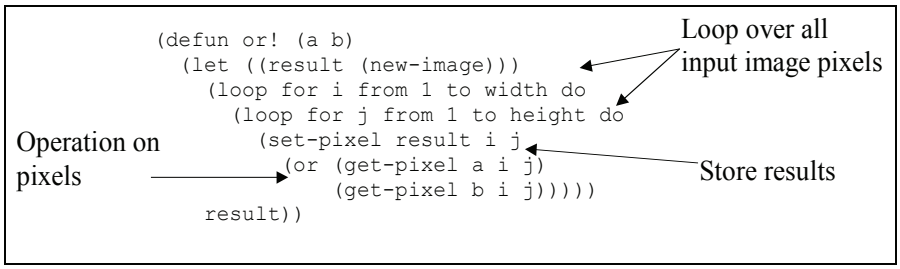

Fig. 3. Definition of or! Filter from [14]

Using similar loop structure, primitive logical filters can be programmed. From these primitive filters, higher level filters, such as `horizontal-edge!`, can be built. The necessary filters to implement the `horizontal-edge!` filter is shown in **Table 1**.

Table 1. Filters Necessary for Horizonal Edge Filter from [14]

functionality	implementation
pixelwise logical operations	written using loop primitive as or!
shift image up, down	written using loop primitive; slightly different loop structure
difference of two images	`(defun remove! (a b)` `(and! a (not! b)))`
pixels at top edge of a region	`(defun top-edge! (a)` `(remove! a (down! a)))`
pixels at bottom edge of a region	`(defun bottom-edge! (a)` `(remove! a (up! a)))`

horizontal edge pixels	`(defun horizontal-edge! (a)`
	`(or! (top-edge! a)`
	`(bottom-edge! a)))`

Functional decomposition

The above implementation of the `horizontal-edge!` has a clean component structure that is easy to understand. The higher level filters are implemented using the more primitive filters. The functional decomposition diagram shown in Fig. 4 below shows the functions that are called when `horizontal-edge!` is called.

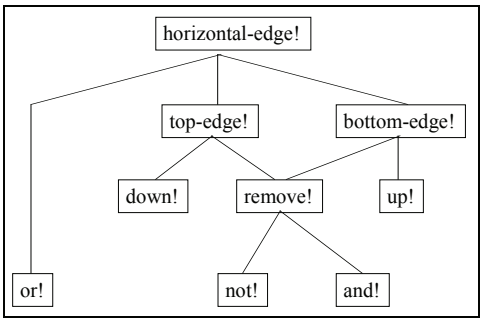

Fig. 4. Functional Decomposition of horizontal-edge! from [14]

This functional decomposition could be the result of a conceptual integration network. In this integration network, each filter function is contained within a mental space. The similarities between the function names create cross-space connections. If the functions are organized spatially similarly to the functional decomposition diagram, the resultant network has similar structural organization as shown in Fig. 5. The difference is that only the function names are shown in the functional decomposition diagram and the LISP syntax and method body are left out. Furthermore, the integration network diagram organizes the linear code into a two dimensional diagram to produce the functional decomposition diagram. Therefore, the integration network shown in Fig. 5 describes how the functional diagram could have been created from the LISP implementation.

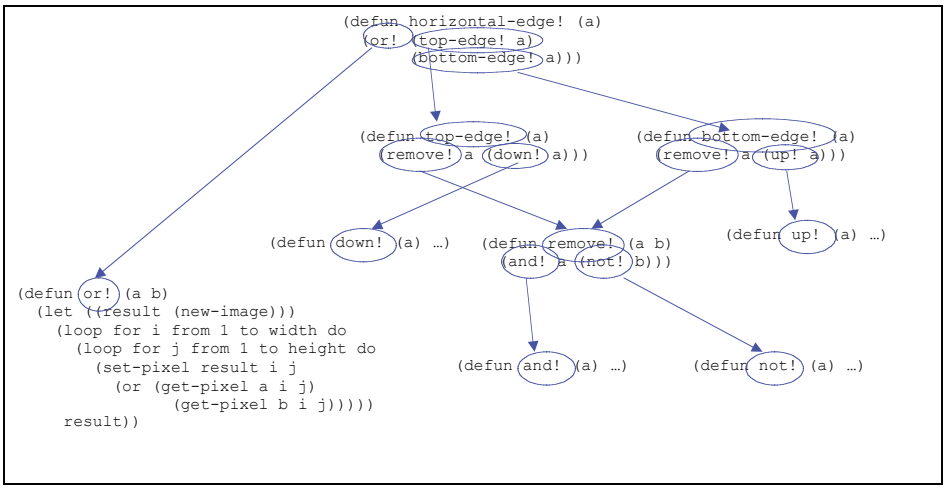

Fig. 5. Function Call Integration Network for horizontal-edge!

There are two things to note from this integration network. First, the structure of the functional decomposition diagram can be created from the LISP code. Another way of looking at it is that some cognitive process took place that produced the

functional decomposition diagram from the LISP implementation. Second, the functional decomposition diagram provides a view of the implementation. This means that information contained within the functional decomposition diagram is present in the implementation and it only needs to be extracted. Therefore, there is no new information inherent in the diagram.

The integration network in Fig. 5 shows how the functional decomposition diagram could have been extracted from the LISP code. However, it does not show how we are able to understand the relationship between the LISP implementation and the diagram. The integration network in Fig. 6, shows how we understand the functional decomposition diagram in relation to the LISP implementation. The similarity between the function names creates the cross-space connection between the particular function definition and the corresponding box. The cross-space connection in this case is Representation, which means that the box with a function name represents the corresponding LISP implementation. The lines between the boxes in the functional decomposition diagram are connected to the cross-space connections between the LISP implementation, which are function calls. Therefore, the understanding of the functional decomposition diagram requires the integration network shown in Fig. 5 and Fig. 6. In essence, the integration network in Fig. 5 is the understanding of function calls and Fig. 6 is the relation between the function calls and the functional decomposition diagram.

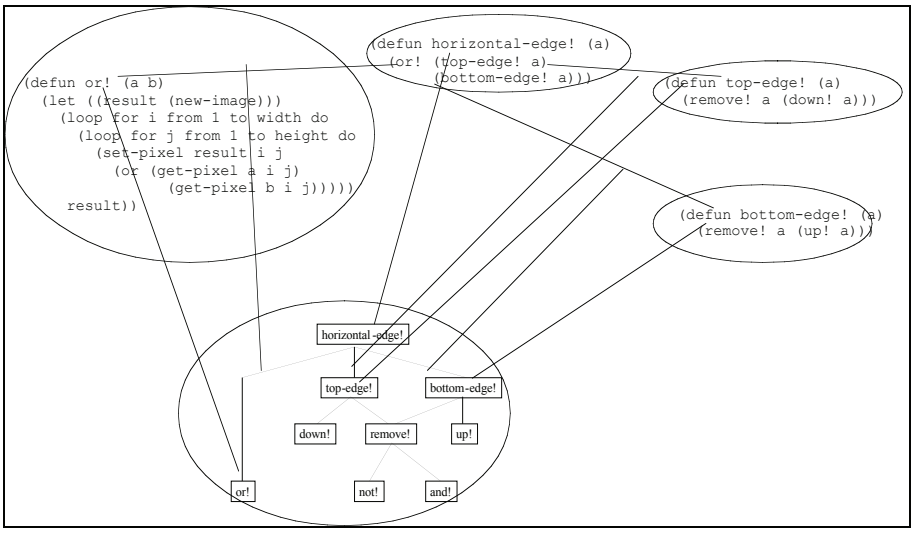

Fig. 6. Integration Network for Understanding the Function Decomposition in Relation to the LISP Implementation of `horizontal-edge!` function (only part of the network is shown)

Data Flow Decomposition

Another way of decomposing the group of functions related to `horizontal-edge!` is to decompose it in terms of the data flow. The dataflow graph from [14] is shown in Fig. 8 left. In order to understand the data flow graph, one needs to create an integration network, in which the original image `a`, which is passed to the `horizontal-edge!` function, is traced during program execution as shown in Fig. 7 left. In CIT parlance, the tracing of the image `a` is an elaboration or a mental simulation. During this elaboration, a blended space, shown on the right in Fig. 7 right, is created to keep track of the operations that are applied to the image `a`. At point (1) in the elaboration, function call to `up!` is projected onto the blended space. The call to `not!` with the result from `up!` is projected to the blended space at point (2) in the elaboration. This process continues until the elaboration stops at the end of the `horizontal-edge!` function and a conceptual structure that is similar to the dataflow graph is in the blended space. The dataflow graph in the blended space is represented as a LISP expression in Fig. 7. However, a diagram could also depict the conceptual process. It is important that the conceptual structure is same between what is constructed in the blended space and the dataflow graph and not the actual form of the representation.

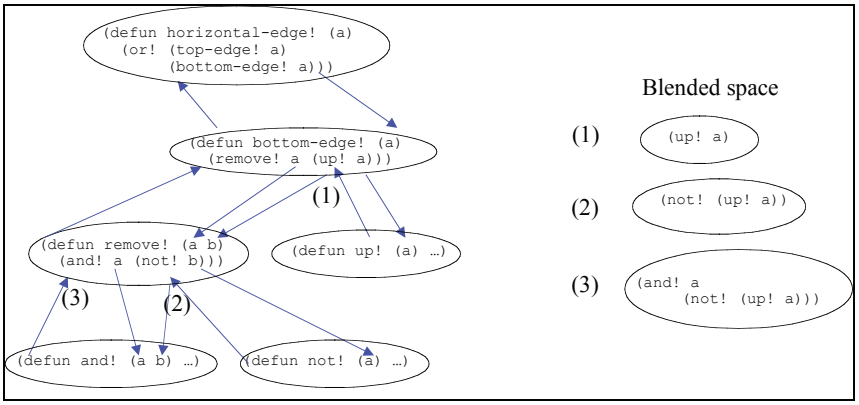

Fig. 7. Integration Network to Understand Data Flow Decomposition (Only the first three steps in constructing the dataflow graph are shown)

In contrast to the functional decomposition diagram, the dataflow graph is much more complex to understand. This is due to the mental simulation or elaboration needed within a blended space to keep track of the projected elements, i.e. the methods with similar loop structure to the `OR` filter, visited during the elaboration. Despite this difference between the two diagrams, the information in the data flow graph is also present in the LISP code, same as the functional decomposition diagram. Each of the boxes and the connections present in the dataflow graph can be related to the LISP code with more effort. The integration network relating the elaboration in Fig. 7 to the dataflow diagram is not shown for space reasons.

Optimization

The implementation of the `horizontal-edge!` function as described above is well modularized and easy to understand. The higher-level filter is implemented using more primitive filters. However, this implementation is inefficient in memory usage as Kiczales et al. point out. There are several intermediate images that are created just to pass the resultant image from one filter to another. This results in frequent memory access over large memory space that lead to poor performance. As Kiczales et al. observe, "The familiar solution to the problem is to take a more global perspective of the program, …". This global perspective is the data flow diagram in Fig. 8 left. From this diagram, it is easy to perceive that the functions `up!`, `down!`, `or!`, `and!`, and `not!` are neighbors in the data flow and potentially, they could be merged. However, only the `or!`, `and!` and `not!` functions share the same loop structure. Therefore, these functions can be merged into one function and use the same loop structure such that several operations are applied to each pixel at the same time. Thus, this new merged implementation would require less memory access and improve performance. The data flow diagram showing the neighboring filters having the same loop structure and the data flow diagram after the functions are merged is shown in Fig. 8.

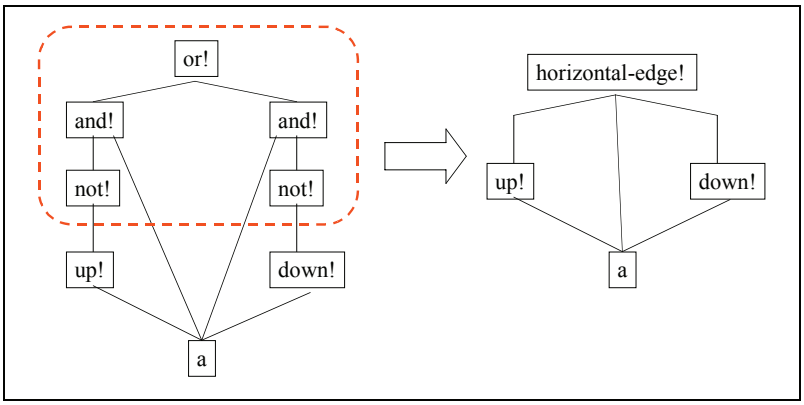

Fig. 8. Data Flow Diagram Before and After Optimization from [14] (dotted box indicates neighboring functions with the same loop structure)

An implementation of the optimization is shown in Fig. 9. The optimized code only creates three intermediate images. The same loop structure is shared between the `or`, `and`, and `not` sub-filters. Notice that this optimized implementation destroys the easy to understand functional decomposition and now it is harder to understand. The functionality of the sub-filters are 'tangled' into the shared loop body. This tangled code leads to maintainability problems: "The tangled code is extremely difficult to maintain, since small changes to the functionality require mentally untangling and then re-tangling it" [14]. Therefore, the understanding of the optimized code requires one to create the integration network that leads to the functional decomposition diagram from the optimized code. Changes to the functionality requires further step of making changes to the unoptimized code and then optimizing the code again.

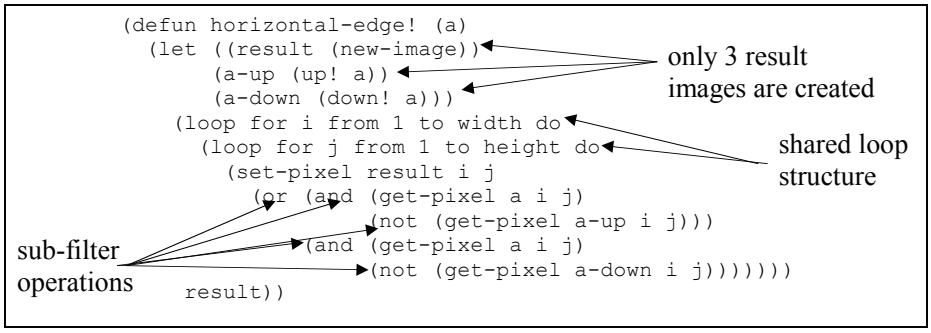

Fig. 9. Optimized `horizontal-edge!` Filter from [14]

The aspect-oriented programming as advocated by [14] solves this problem by providing linguistic facilities to automate this optimization process. The linguistic features are directives to identify and merge similar loop structure in the program. They argue that this approach allows the programmer to maintain the original functional decomposition while providing optimized code that minimizes memory access. However, the programmer must still mentally tangle the code in order to understand how the directives change the code. The solution to this cognitive operation of tangling code could be obtained if one notices the role of representations in the form of functional decomposition diagram, data flow diagram, and LISP code in the image filter example. It is evident that these representations are interrelated through the conceptual integration networks as described above. These conceptual integration networks in which the function definitions are inputs can characterize the creation and understanding of the functional decomposition and data flow diagrams. Therefore, it would seem likely that if these cognitive processes of tangling and untangling are offloaded onto the computer, it could potentially improve the understanding and development of software.

Relationship Between Representations

In the previous section, the conceptual integration networks were used to describe how we are able to understand the image filter optimization example from [14]. These integration networks characterize not only the cognition required in understanding the various representations, it also provides a description of how these representations could have been constructed. This cognitive semantic analysis exposes several properties about the representations and their role in software development. These properties are outlined in the conceptual integration theory as three properties of integration networks: global insight, unpacking and synchronization. These three properties taken within the context of distributed cognition and cognitive offloading suggest that a representation-oriented approach to software development would be productive.

The first notion in conceptual integration theory that is relevant in understanding the role of representations in the image filter optimization example is global insight. *Global insight* is the notion that in order to gain a comprehensive understanding, one has to know the details and the global view of a particular subject. In this case, one has to know the LISP implementation and the functional decomposition diagram at the same time. The functional decomposition diagram provides an overview of how the functions are connected, while the LISP implementation describes what each of the functions contribute towards the overall functionality of the `horizontal-edge!` function. Therefore, the two diagrams provide global insight into the LISP implementation. Also, the data flow diagram provides global insight, which facilitates understanding of the optimized code and characterizes how the optimization could have been solved originally.

The second relevant property of integration networks is the *unpacking* principle. It states that the integration network tends to be constructed in such a way that elements contained within the blended space are sufficient to reconstruct the original integration network that led to the blended space. In this case, the integration networks in characterizing how the two diagrams could have been produced could be reconstructed from the diagrams alone. In essence, one should be able to write the LISP implementation from the functional decomposition diagram or the data flow diagram. Intuitively, this is plausible, since software could be developed in such a way that organization of the functionality is designed first then implemented in LISP.

The last relevant property of integration networks is the propagation of changes. In the process of constructing meaning, elements in the blended space could interact in such a way that new elements could be brought forth into input spaces and vice versa. In the context of representations, some change in one representation could propagate to other representations. When the original image filter implementation is optimized, the corresponding functional decomposition and data flow diagrams also reflect this change in the LISP code. Alternative view is that the change in the data flow diagram is propagated to the LISP code and the functional decomposition diagram. Either way, the structural similarity between the diagrams and the LISP code is preserved in this synchronization process.

Representation Oriented Software Development

The three properties, global insight, unpacking and synchronization, of integration networks observed in the understanding of the three representations, functional decomposition diagram, data flow diagram, and LISP code, show that software development is a multi-representation activity. In a sense, software includes these representations and others such as code comments, architecture diagrams, specifications, requirements, etc. and developers create, manipulate and make sense of these representations. From the cognitive semantic analysis presented above, these cognitive activities of creating, manipulating and understanding representations can be characterized by the conceptual integration networks. If role of conceptual integration networks in understanding representations is taken within the context of distributed cognition, a new approach to software development emerges. I call this approach representation-oriented software development (ROSD).

In ROSD, the central issue is the design, manipulation and maintenance of representations for optimizing the cognitive system. The cognitive system includes representations, humans, computers and other artifacts involved in software engineering. All of these elements interact and coordinate in producing software, similar to the way a crew of a ship coordinate through artifacts to navigate a ship. This view of software engineer reconceptualizes the problem of optimizing software development to the problem of optimizing the cognitive system through redistribution of cognition. Since computers tend to increase in speed much quicker than humans, it would be logical to find opportunities to offload human cognition onto the computer. The conceptual integration theory provides a methodology for finding such opportunities through its integration networks. These networks characterize the cognitive processes required in producing and manipulating representations. Therefore, offloading this cognition would entail that software tools should be developed to support the developers in designing, manipulating and maintaining these representations.

Conclusions

The cognitive semantic analysis of understanding the image filter optimization from [14] within the context of distributed cognition suggests a representation-oriented software development approach. This approach puts the representations in software and offloading cognition in processing these representations to the computer as its main conern. In relation to AOSD, ROSD is more specific in a sense that it bases its theoretical framework on distributed cognition and conceptual integration theory. This relationship between AOSD and ROSD is reflected in the image filter optimization example when the aspect-oriented programming solution in [14] is conceptualized as offloading cognition required to optimize the image filter onto the computer. Therefore, the linguistic features proposed by [14] are prompts for executing the offloaded human cognition.

This work is complementary to the work done by other cognitive scientists in applying distributed cognition to issues in HCI [18], collaborative working [19], naturalistic programming [20], and computational aspects of figurative language [24]. The main difference is the application of cognitive semantic theories to find opportunities for offloading cognition onto the computer. This approach is novel and there are many opportunities for further development. For instance, the other aspects of distributed cognition, such as material anchors [21] could be explored, or apply other cognitive semantic theories such as idealized cognitive models [22] to analyze and offload cognitive structures and processes onto the computer. Since idealized cognitive models are shown to be observed in classical mechanics [23], it is highly likely that software tools that offload cognition in understanding these concepts, problems and solutions would facilitate providing software tools for science, engineering and other fields.

References

1. Norman, D.A.: The Design of Everyday Things. Basic Books, New York (2002)
2. Preece, J., Rogers, Y., Sharp H.: Interaction Design: Beyond human-computer interaction. J. Wiley & Sons, New York (2002)
3. Sommerville, I.: Software Engineering. 5th Ed. Workingham: Addison-Wesley, (1995)
4. Hutchins, E.: Cognition in the Wild. Cambridge, Mass., MIT Press. (1995)
5. Fauconnier, G., Turner, M.: Conceptual integration networks. Cognitive Science 22 (1996) 133-187
6. Fauconnier, G., Turner, M.: Compression and global insight. Cognitive Linguistics 11, 3/4 (2000) 383-304
7. Fauconnier, G., Turner, M.: The Way We Think: Conceptual blending and mind's hidden complexities. Basic Books, New York (2002)
8. Filman, R.E., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Boston: Addison-Wesley, (2004)
9. Elrad, T., Filman, R.E., Bader, A.: Aspect-oriented programming: Introduction. Comm. Of the ACM. 44 (10), (2001) 29-32
10. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Girswold, W.: Getting started with ASPECTJ. Comm. of the ACM, 44 (10), (2001) 59-65
11. Bergmans, L., Aksit, M.: Composing crosscutting concerns using composition filters. Comm. of the ACM, 44 (10), (2001) 51-57
12. Lieberherr, K.: Controlling the complexity of software designs. Proc. of the 26th International Conference on Software Engineering. (2004) 2 - 11
13. Ossher, H., Tarr, P.,: Hyper/J: Multi-dimensional separation of concerns for java. Proc. of the 22nd International Conference on Software Engineering. (2000) 734-737
14. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In Proc. Of the Europ. Conf. on OOP. Springer-Verlag, Findland (1997)
15. Croft, W., Cruse, D.A.: Cognitive Linguistics. Cambridge University Press, Cambridge (2004)
16. Evans, V., Green, M.: Cognitive Linguistics: An introduction. Edinburg University Press, Edinburg (Forthcoming)
17. Janssen, T., Redeker, G. (Eds.): Cognitive Linguistics: Foundations, scope, and methodology. Mouton de Gruyter Berlin (1999)
18. Hollan, J., Hutchins, E., Kirsh, D.: Distributed cognition: Toward a new foundation for human-computer interaction. ACM Trans. On Comp.-Human Interaction (TOCHI), 7 (2), (2000) 174-196
19. Rogers, Y., Ellis, J.: Distributed cognition: an alternative framework for analyzing and explaining collaborative working. Journal of Information Technology, 9 (2), (1994), 119-128
20. Lopes, C.V., Dourish, P., Lorenz, D. H., Lieberherr, K.: Beyond AOP: Toward naturalistic programming. ACM SIGPLAN Notices, 38 (12), (2003) 34-43
21. Hutchins, E.: Material anchors for conceptual blends. Journal of Pragmatics (in press), http://hci.ucsd.edu/lab/hci_papers/EH2004-1.pdf
22. Lakoff, G.: Women, Fire and Dangerous Things. Chicago London: Cambridge University Press. (1987)
23. Giere, R. N.: The cognitive structure of scientific theories. Philosophy of Science 61 (2), 276-296
24. Barnden, J. A., Glasbey, S., Lee, M., Wallington, A.: Domain-transcending mappings in a system for metaphorical reasoning. In Proc. of the 11th Conference of the European Chapter of the Association for Computational Linguistics: 57-62