UC Merced

Proceedings of the Annual Meeting of the Cognitive Science Society

Title

Can We Extend the Reverse Cohesion Effect to Programming Contexts?

Permalink

https://escholarship.org/uc/item/9wm1d687

Journal

Proceedings of the Annual Meeting of the Cognitive Science Society, 47(0)

Authors

Harsch, Rina Miyata
Bye, Jeffrey K.
Rus, Vasile
et al.

Publication Date

2025

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, available at https://creativecommons.org/licenses/by/4.0/

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Can We Extend the Reverse Cohesion Effect to Programming Contexts?

Rina Miyata Harsch (harsch@umn.edu)
University of Minnesota, Department of Educational Psychology, 56 East River Road, Minneapolis, MN 55455

Jeffrey K. Bye (jkbye@csudh.edu)
California State University, Dominguez Hills, Department of Psychology, 1000 E. Victoria Street, Carson, CA 90747-0001

Vasile Rus (vrus@memphis.edu)
Dunn Hall 375, University of Memphis, Memphis, TN 38152-3240

Panayiota Kendeou (kend0040@umn.edu)
University of Minnesota, Department of Educational Psychology, 56 East River Road, Minneapolis, MN 55455

Abstract

Existing research has drawn parallels from the comprehension of text to the comprehension of source code. In this study, we attempt to develop this analogy by positing and testing a notion of *code cohesion*, analogous to text cohesion. We also attempt to extend a known effect in text comprehension research, the *reverse cohesion effect*, to code contexts. Our findings provide some corroboration for code cohesion, but fail to find robust evidence for a reverse cohesion effect. This reinforces similarities between text and code comprehension but also suggests that everyday comprehension processes of code and text might differ in meaningful ways.

Keywords: code comprehension; text comprehension; cohesion; reverse cohesion effect

A critical skill for programmers, whether novices or experts, is reading code. Reading code is the most time-consuming part of software maintenance (Rugaber, 2000). Thus, one important goal of computer science education is to cultivate code comprehension skills. In service of this goal, researchers have developed a theory of source code comprehension that draws an analogy from text to code.

Code comprehension involves the construction of a mental representation of code with three dimensions: the intended *purpose* of the code including background information about its domain, the *procedure* or logical steps that achieve the goal, and the *behavior* such as the functions or data structures involved (Schulte et al., 2008). This is parallel to the three-dimensional representation constructed during text comprehension (van Dijk & Kintsch, 1983). This representation of code is constructed through passive, bottom-up processes and strategic, top-down processes (Letovsky, 1986), as it is in text comprehension (Kendeou & O'Brien, 2018; McNamara & Magliano, 2009).

The Current Study

Our purpose in the present study is to further develop the analogy from text comprehension to code comprehension. To do so, we posit and empirically test a construct called *code cohesion*, conceptualized by analogy from *text cohesion*. Text cohesion refers to the extent to which a text's relations, concepts, and ideas are made explicit (O'Reilly & McNamara, 2007). Text cohesion affects comprehension; cohesive text is theoretically easier to comprehend because it facilitates the bottom-up and top-down processes recruited during comprehension (McNamara & Magliano, 2009).

By analogy, code cohesion refers to the extent to which the purpose, procedure, and behavior of a code is made explicit in the code and comments. In this study, we design and test manipulations of code cohesion. Efforts have been made to identify code features that improve comprehension in the study of an adjacent construct, code readability (Scalabrino et al., 2018). This work generally tests metrics based on features of existing code, rather than systematically manipulating these features. Furthermore, readability metrics are often inspired by commonly accepted best practices; while thoughtful, they are not usually driven by code comprehension theory. Drawing instead on theory might lead to the identification of code features not previously considered. We anticipate that our manipulations of code cohesion will improve code comprehension. Specifically, people should demonstrate better code comprehension by accurately describing its purpose and predicting its output when the code is cohesive.

To further test the validity of this construct, we also attempt to extend the *reverse cohesion effect* from text to code contexts. The reverse cohesion effect (McNamara & Kintsch, 1996) refers to the finding that readers who have low levels of prior knowledge on a topic comprehend texts about that topic more accurately when those texts are cohesive than when they are not cohesive; however, the effect of cohesion on comprehension disappears or is reversed for readers with higher levels of prior topic knowledge (O'Reilly & McNamara, 2007). This reverse cohesion effect has been conceptualized as a 'desirable difficulty' determined by learner characteristics (McDaniel & Butler, 2011). We anticipate that we might find a *reverse code cohesion effect*, in which programming novices better comprehend cohesive than incohesive code, but the benefit of cohesion disappears or even reverses for experts.

Experiment 1: Validation

In this experiment, we aim to provide initial validation for the construct of code cohesion. We tested whether 8 novel

In D. Barner, N.R. Bramley, A. Ruggeri and C.M. Walker (Eds.), *Proceedings of the 47th Annual Conference of the Cognitive Science Society* ©2025 the author(s). This work is licensed under a Creative Commons Attribution 4.0 International License (CC BY).

manipulations of code cohesion empirically resulted in code that was perceived by programmers to be more cohesive. We posed the following research questions: 1) Is high cohesion code associated with greater perceived ease in identifying the purpose, procedure, and behavior?; and 2) Which manipulations affect perceived ease the most?

Materials

The experiment was programmed using custom plugins in jsPsych (de Leeuw et al., 2023) and run online.

Code Stimuli We developed four different code stimuli templates (hereby, code loops) for this project. All code loops were short and included one loop (a block of code that is run repeatedly). They were adapted from previous examples used in a research study with introductory computer science students (Oli et al., 2024). The loops varied in terms of their purpose, the procedure, and behavior.

Manipulations of Code Cohesion We tested seven individual manipulations of code cohesion inspired by text cohesion research (Kintsch & van Dijk, 1978), code readability research (Scalabrino et al., 2018), and programming style guides, as well as an eighth condition combining all seven individual features. To systematically and orthogonally apply these manipulations, we first made neutral-cohesion versions of all seven manipulations for each of the four code loops. Individual manipulations were then made on the neutral version of the code, ensuring that the code was high or low in the intended type of cohesion, but neutral in terms of the other types (except for the Super High/Low combination condition).

Syntactic Manipulations of code cohesion directly changed the syntax of the codes in ways that theoretically facilitate bottom-up comprehension processes. *Argument Overlap* refers to the extent to which identifiers (e.g., a variable name) appeared in consecutive lines. More overlap was deemed more cohesive. *Lengthiness* refers to the extent to which the lines of code are complex, nested, and few (low lengthiness) or simple but more numerous (high lengthiness, which is more cohesive).

Semantic Manipulations of code cohesion involved manipulating names and terms, not syntax. High semantic cohesion theoretically facilitates comprehension through top-down processes. *Comment Helpfulness* refers to the extent to which the comments include information about the three dimensions of the code. *Identifier Meaningfulness* refers to the informativeness of identifier terms. For example, the identifier `fruitsDictionary` informs us of the code's purpose and behavior, but the identifier `a` does not.

Aesthetic Manipulations of code cohesion involved changing the visual features, but not the syntactic or semantic features. To manipulate *Syntax Highlighting*, we varied the font color and formatting of parts of the code. High syntax highlighting used the same default highlighting scheme as the popular Jupyter Notebook environment; low syntax highlighting was uniformly in black font. *Vertical Spacing* was manipulated by adding or removing blank lines into the code to segment the code into logical components (analogous to paragraphs in text), with more breaks in high vertical spacing and none in low vertical spacing codes.

The ***Pragmatic Manipulation*** of code cohesion was *Signaling the Beacon*. This refers to the inclusion or exclusion of a comment that draws attention to a code's "beacon", or crucial segment of code that can be informative regarding the code's purpose and program (Brooks, 1983). High signaling the beacon included the comment `## IMPORTANT`; codes without this comment were low cohesion. Theoretically, signalling the beacon might facilitate code skimming, sometimes used by skilled programmers for comprehension (Wiedenbeck, 1986).

Finally, in the ***Super*** high and low cohesion manipulations, we implemented all seven high or all seven low manipulations at once.

Set Design

In total, there were 64 unique stimuli: high and low cohesion versions of the eight different manipulations for each of the four code loops. We then devised 32 different stimuli sets, each containing 16 stimuli: high and low versions of two different manipulations for each of the four code loops. Each set had high and low versions of all eight types of cohesion across the set. This allowed us to counterbalance the order, pairing, and code loops in which each manipulation appeared, across participants.

Study Design

We used a 4 (different code loops) by 2 (low vs. high cohesion level) by 8 (types of cohesion manipulation) by 2 (novice vs. expert programmers) design. Code loop, cohesion level, and cohesion manipulation varied within participants, while programming expertise varied between participants. For control purposes, we randomized the orders of the four code loops within participants, the order of the two manipulations for each code (within participant, within code loop), and the order of the two cohesion levels for each manipulation (within participant, within code stimulus, within manipulation).

Participants

We recruited participants from the Prolific crowdsourcing platform. First, we screened participants who were studying computer science (CS) or working in software or related industries. Screener participants were paid $0.50 for less than two minutes. Of those participants, we identified novice and expert programmers. Novices were defined as those who had two years or less of programming and Python experience and either took fewer than ten CS courses or worked in the industry for less than two years. Expert programmers had five or more years of programming and Python experience and either took more than three CS courses or had two or more years of industry experience. As no explicit thresholds have been established for programming expertise, we loosely followed previous research (Dorn, 2012; Wiese et al., 2019).

We recruited participants from these prescreened samples until we reached 65 novices and 64 experts. These participants were paid $7.50 for 30 minutes of their time. Their average age was 30.70 years (*SD* = 10.02). 29 were assigned female at birth, and 99 were assigned male at birth. Regarding race and ethnicity, 32 participants identified as Asian, 13 as Black, 72 as White, 5 as multiracial, and 1 indicated Other.

Procedure

Participants accessed the study through Prolific. After consenting, they received instructions that they would be rating code in Python. On each page, they would see four versions of the same script, and would be asked questions relating to the script's purpose, procedure, behavior, and readability. Purpose was defined to participants as "why the program was written, i.e., what the user's goal is", procedure as "the control flow of the program, i.e., in what order things happen", and behavior as "the data structures of the program, i.e., how data is stored and changed." Readability was defined as "how easy it is to understand the program just by reading it (without running the program)."

Then, participants were presented with four versions of the same script per page. The top two versions were high and low cohesion versions of one manipulation, and the bottom two were high and low cohesion versions of another manipulation. First, participants were asked questions about the program's purpose and output, to ensure they attended to the stimuli. Then, they rated each version of the script in terms of how easy or difficult it was to identify its purpose, procedure, and behavior on a Likert-type scale from 1 (extremely difficult) to 6 (extremely easy). On the next page, they rated the readability of each version of the script on a Likert-type scale from 1 (extremely unreadable) to 6 (extremely readable). Two attention checks were included.

Results

For each version of a code loop, we computed a mean ease metric, which captures how easy each participant reported it was to identify the key components of the code on average. This was computed by taking the average of their ratings of ease of identifying the purpose, procedure, and behavior for that code stimulus version, per participant. Figure 1 shows the difference in mean ease ratings between high and low cohesion codes for each manipulation, from highest difference to lowest, left to right. Descriptives of mean ease ratings (1-6 scale), are given in Table 1.

Using paired sample *t*-tests, we found that participants' mean ease ratings differed between the high and low cohesion code loops for all manipulations but Signalling the Beacon (Table 1). Of those manipulations, mean ease ratings were higher for the high cohesion version than the low cohesion version, except for Lengthiness, for which the low cohesion version was rated easier. The Super manipulation resulted in the largest difference in mean ease ratings between high and low cohesion versions.

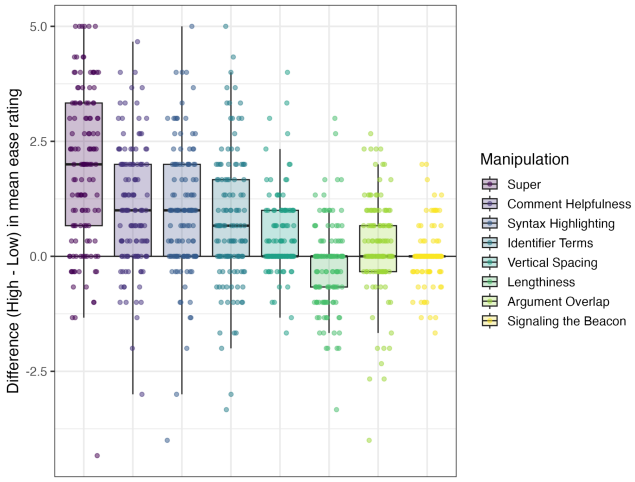


Figure 1: Difference in Mean Ease Ratings between High and Low Cohesion, by Manipulation

Table 1: Mean Ease Ratings for Each Manipulation by Cohesion Level

	Cohesion Level			
	High	Low		
Manipulate	*M*(*SD*)	*M*(*SD*)	*t*(128)	*p*
Super	5.04(0.46)	3.17(0.59)	-12.44	<.001
Comments	5.22(0.38)	4.15(0.58)	-9.37	<.001
Highlight	5.29(0.35)	4.34(0.49)	-8.01	<.001
Identifiers	5.02(0.41)	4.28(0.54)	-5.94	<.001
Spacing	5.12(0.33)	4.72(0.41)	-5.46	<.001
Length	4.83(0.47)	5.04(0.39)	2.68	.008
Overlap	5.09(0.41)	4.87(0.47)	-2.44	.016
Beacon	5.07(0.34)	5.05(0.39)	-0.36	.721

Experiment 2: Reverse Cohesion

The results of Experiment 1 provided novel validity evidence that certain manipulations of code cohesion are judged by readers to be easier to comprehend. The Super cohesion manipulation showed the largest difference between the high and low conditions. In Experiment 2, we examined whether Super high or low levels of code cohesion affect participants' actual comprehension of the code and whether we observe a reverse code cohesion effect for the first time.

Given the parallels between text and code comprehension, we anticipated finding a reverse cohesion effect for code. Specifically, we hypothesized that we would find evidence that readers' comprehension of code is predicted by the interaction between programming experience and code cohesion, such that less experienced programmers benefit from cohesive code, but this benefit reduces or reverses for participants with higher levels of programming knowledge.

Materials

The experiment was programmed using custom plugins in jsPsych (de Leeuw et al., 2023) and run online.

Code Stimuli The same four code loops validated in Experiment 1 were used in their Super high and low forms. We anticipated that participants might assume that the code loops were written correctly and would execute the intended purpose of the code. If so, they might not effortfully attend to the full code while reading, undermining the cohesion manipulation. To preempt this problem, we devised a *misleading* version of each code stimulus. Misleading versions were identical to the correct versions except for part of one line that caused it to not function as intended. Misleading versions would compile without a runtime error but would not print the intended output.

We crossed code cohesion (high/low) and correctness (correct/misleading) for each of the four code loops, yielding a total of 16 stimuli in the master stimulus set. To construct four different sets, we distributed the 16 code loops such that each set contained all four code loops and all four combinations of cohesion and correctness, with all features counterbalanced across sets (and thus, participants). Stimulus order was randomly shuffled for each participant.

Programming Experience To approximate participants' programming experience, we included the revised version of the Second CS1 Revised aptitude test (SCS1R; Bockmon et al., 2019). The SCS1R is a 9-item multiple choice questionnaire on core concepts in introductory CS courses. Each item is about a computer science concept or includes an example of code. Participants are prompted to answer the question or identify the code's output, depending on the item type. While the SCS1R items were originally written in pseudocode, we translated all items to Python.

Participants' total accuracy was used as a measure of their programming experience. For this sample, participants' mean score was 5.60 (SD = 2.20). This correlated to a low to moderate degree with other measures of programming experience: Python experience (Pearson's r = 0.34) and their reported number of CS courses (Pearson's r = 0.26).

Study Design

This study used a 2 (cohesion: Super high vs. Super low) by 2 (correctness: correct vs. misleading) within-participants design, with programming experience as a continuous, between-participants variable.

Participants

We recruited 177 participants from CS courses at a large Midwestern U.S. university. They received extra credit in their course. Thirteen were removed from the analyses after failing an attention check, yielding a final sample of n = 164. Participants' mean age was 20.37 years (SD = 3.38). Regarding gender, 110 identified as men, 47 as women, 1 as non-binary or gender non-conforming, and 6 preferred not to say. In terms of race and ethnicity, 2 identified as Arab, 58 as Asian, 17 as Black, 3 as Native American or American Indian, 3 as Hispanic or Latine, 73 as White, 6 as Mixed, and 2 declined to respond. Their Python experience ranged from 1 month to 7 years, $M(SD)$ = 1.77(1.88) years.

Procedure

After indicating consent, participants were randomly assigned to one of the four sets. They were instructed that they would be reading codes written in Python, predicting their outputs, and describing their purposes. They were also instructed to read the codes carefully, as some would not run as intended. Then, they were presented with the first code. With the code still on the screen, participants were asked two open-ended questions: "When this code is run in Python, what is the exact output that the code will produce?" and "What is the intended purpose of this program? In other words, what was the programmer's goal for this code? Please be as specific as possible." The same questions were given for the three other codes. Next, the participant completed the SCS1R, answered questions about their programming experience, and completed the demographics form.

Results

The first author scored the responses to the output and purpose questions as 0 or 1. Minor errors were permitted for the output item. Correct answers for the purpose items were established by computer science experts who read each code snippet. For the analysis, we constructed logistic mixed effects models for each dependent measure of code comprehension (output accuracy and purpose accuracy, 0 or 1) with fixed effects of correctness (correct vs. misleading), cohesion level (super high vs. super low), programming experience (SCS1R, out of 9), and the cohesion-by-SCS1R interaction. All predictors were centered; binary predictors (correctness and cohesion level) were coded as -0.5 and 0.5, and SCS1R was Z-scored. We included random effects of participant and code loop.

Overall Analyses For output accuracy, we did not find evidence for a reverse code cohesion effect (Figure 2).

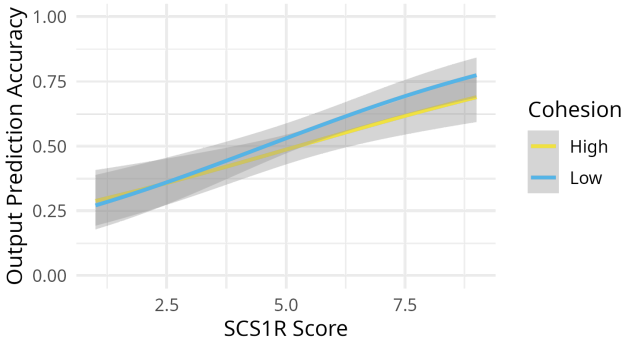


Figure 2: Output accuracy based on programming knowledge (SCS1R), separated by cohesion level. Curves represent fits from logistic mixed effects models.

We found no cohesion-by-programming experience interaction effect (B = -0.24, z = -1.27, p = .204). We also did not find a main effect of cohesion (B = -0.31, z = -1.63, p = .103). However, we found strong evidence for the

effects of correctness (B = 1.50, z = 7.38, p < .001). The odds of correctly predicting the output of code that functioned as intended were 4.47 times that of misleading code, on average, controlling for other factors. Additionally, we found an effect of programming experience (B = 0.71, z = 6.25, p < .001). Controlling for other variables, each 1-point increase in SCS1R was associated with a 2.04-fold increase in the odds of correct output prediction, on average.

Similarly, we did not find a reverse cohesion effect on comprehension of code purpose, though the data show tendencies toward one (Figure 3). We return to this below.

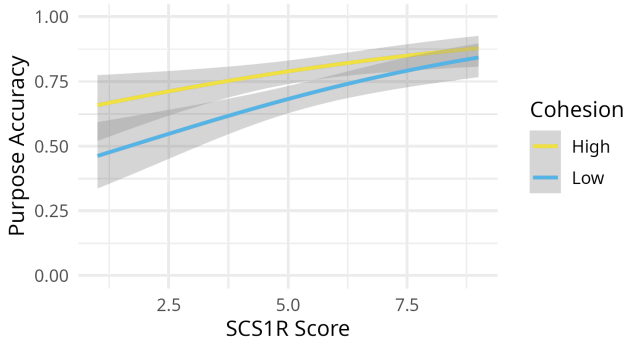


Figure 3: Purpose accuracy based on programming knowledge (SCS1R), separated by cohesion level

Purpose Comprehension Given that understanding the code's purpose might facilitate the comprehension of its program and behavior and thus its output, we explored the relation between these two measures. Specifically, we sought to explore whether participants' purpose comprehension predicted their output prediction, as well as whether or not controlling for purpose comprehension would clarify the effect of cohesion on output prediction. To do this, we again used a logistic mixed effects model with output accuracy as the outcome, and with cohesion, programming experience, their interaction, correctness, and purpose accuracy as predictors. We included the same random effects.

The main effects of correctness (B = 1.47, z = 7.27, p < .001) and programming experience persisted (B = 0.65, z = 5.83, p < .001). Reading correct code (instead of misleading code) increased participants' odds of accurate output prediction 4.34-fold. Additionally, each 1-point increase in SCS1R was associated with a 1.92-fold increase in the odds of accurate output prediction.

The results also clarified the complex role of code cohesion in code comprehension. We found a main effect of purpose accuracy (B = 0.67, z = 2.86, p = .004), suggesting that on average, participants were 1.95 times as likely to correctly predict a code's output if they correctly described its purpose, controlling for other variables. Then, controlling for participants' purpose accuracy, we found an effect of cohesion, which was not found in the previous analysis of output accuracy (B = -0.38, z = -1.98, p = .048). When participants read incohesive code, their odds of accurate output prediction increased by a factor of 1.45. Counter to our predictions, code cohesion reduced the odds of accurate output prediction, controlling for purpose accuracy. No reverse cohesion effect was observed, (B = -0.21, z = -1.08, p = .282).

Misleading Stimuli To further explore the roles of purpose accuracy and cohesion in output accuracy, we explored patterns separately for correct and misleading stimuli. We anticipated different results based on correctness because when code loops function as intended, participants can correctly predict the output just from understanding the program's purpose and inferring what it *should* output. However, when they are "misleading", inferring output from the code's purpose will lead to errors; to respond correctly, they need to discover that the code makes a mistake by examining the code's program and behavior. We ran parallel analyses without correctness as a predictor on the two halves of the dataset.

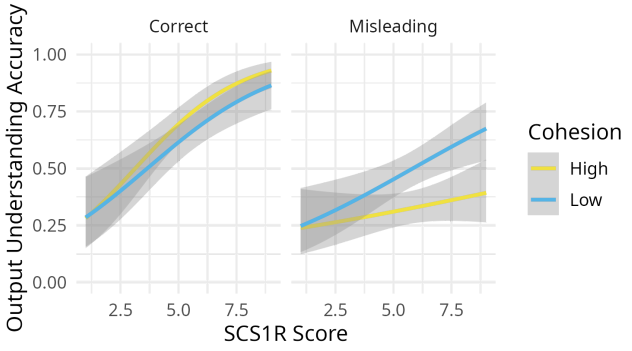


Figure 4: Output accuracy for correct and misleading code loops based on programming knowledge (SCS1R), separated by cohesion level

As we anticipated, we found some similar and some different patterns in the results for correct and misleading codes (Figure 4). For both correct and misleading codes, output accuracy was predicted by programming experience, though more so for the correct codes (B = 0.92, z = 5.14, Bonferroni adjusted p < .001) than the misleading ones (B = 0.35, z = 2.39, adjusted p = .034). Each 1-point increase in SCS1R score corresponded to a 2.52-fold and 1.42-fold increase in the odds of correctly predicting the codes' outputs for the correct and misleading stimuli, respectively.

However, the patterns in the effects of purpose accuracy and cohesion on output accuracy differed between correct and misleading codes. Participants were 2.51 times more likely to correctly predict the output of a correct code when they also correctly described its purpose (B = 0.92, z = 2.75, adjusted p = .012). For correct codes, cohesion did not affect output accuracy (B = 0.15, z = 0.52, adjusted p = 1.000). On the other hand, participants' accuracy in their description of the misleading code's purpose did not predict their accuracy in predicting its output (B = 0.47, z = 1.46, adjusted p =

.289). Instead, it was negatively predicted by the cohesion of the code (B = -0.78, z = -2.89, adjusted p = .008). That is, participants' odds of correctly predicting the output of misleading code was increased for *incohesive* codes by a factor of 2.18, compared to when it was cohesive.

We did not find a reverse code cohesion effect for correct or misleading codes. Although Figure 4 shows a potential trend toward a reverse cohesion effect of misleading stimuli, no statistical evidence was found for an interaction effect between cohesion and SCS1R, for neither correct code loops (B = 0.12, z = 0.41, adjusted p = 1.00), nor misleading code loops (B = -0.34, z = -1.27, adjusted p = 0.41).

General Discussion

In this study, we experimentally tested the analogy from text comprehension to code comprehension. We developed the notion of code cohesion and several ways to manipulate it. We validated that some of these manipulations were perceived to be easier to comprehend. Then, drawing on text comprehension research about the reverse cohesion effect, we investigated the roles of code cohesion and its interaction with programming experience on code comprehension (i.e. the reverse code cohesion effect).

Our findings suggest that the analogy between text and code comprehension might not be so straightforward. Code comprehension might differ meaningfully from text comprehension, and cohesion's role in code comprehension might be more complex than we anticipated. We did not find the hypothesized reverse code cohesion effect. Instead, we found differing patterns between two components of code comprehension (i.e., output prediction and purpose description) and depending on whether the code functioned as intended (i.e., was correct or misleading). Our exploratory analyses suggest that increased code cohesion might facilitate understanding of the purpose of code; however, understanding of the logical flow and behavior of the code might instead be facilitated by reduced code cohesion. We discuss these results in more detail next.

First, we did not find support for our hypotheses that code cohesion would improve comprehension and that this improvement would depend on the extent of programming experience (i.e., a reverse cohesion effect). Our initial analyses found an effect of cohesion on purpose description, but not on output accuracy. They also provide no evidence for a reverse cohesion effect for either dependent variable. These results thus contrast with the research in the field of text comprehension (McNamara & Kintsch, 1996).

Furthermore, our results suggest that code cohesion does not straightforwardly influence code comprehension. In our study, code cohesion improved understanding of a code's purpose, but, according to our exploratory analysis, it *hindered* understanding of a code's output when controlling for understanding of the purpose. These results suggest that mental representations of code have at least two dimensions: the code's purpose and its logical flow and behavior. However, it calls into question the extent of the analogy from text comprehension to code comprehension.

One key difference between text and code comprehension might be the reliance on the computer during code comprehension. According to our exploratory analyses, the effects of cohesion on code comprehension seems to depend on whether the code functioned as intended. These results might suggest that readers are using the cohesion of the code to comprehend the program's purpose and, based on the assumption that it might run as intended, further infer the program's output. This strategy would be effective only if the program runs as intended.

Programmers might employ these strategies if they frequently run code as they read. Doing so has benefits, as the program might throw error messages if the code cannot be executed or output something unexpected if the code does not run as intended. Programmers might run code and try to diagnose any errors or bugs by comprehending the code's output, rather than reading the code carefully, as part of their normal, everyday code comprehension processes. There is no obvious analogue to this in text comprehension, aside from perhaps reading a text to a friend to see if they catch any errors, which is generally not considered part of normal text comprehension processes. The differences between our findings and those found in text comprehension research might be due in part to fundamental differences between everyday comprehension of text and code.

Two potential reasons we did not find a reverse cohesion effect in this study are 1) a limited sample size; and 2) limited variability in programming experience in our sample. Our sample consisted of students from our university, primarily students in their first CS course at the university level. The SCS1R scores from the sample ranged from 1 to 9, almost the entire range of the SCS1R. However, this is a measure of introductory CS concept knowledge, and might not provide the variability needed to capture a reverse cohesion effect. Thus, future work should investigate our research questions again with a sample of more diverse programming experience or using a measure with more item discrimination. We are currently following up with an additional study with the same procedure as the second experiment, but with a screened sample of programming novices and experts on Prolific.

References

Bockmon, R., Cooper, S., Gratch, J., & Dorodchi, M. (2019). (Re)Validating Cognitive Introductory Computing Instruments. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 552–557. https://doi.org/10.1145/3287324.3287372

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, *18*(6), 543–554. https://doi.org/10.1016/S0020-7373(83)80031-5

Dorn, J. (2012). *A general software readability model* Master's Thesis, University of Virginia, Charlottesville, Virginia.

Kendeou, P., & O'Brien, E. J. (2018). Reading comprehension theories: A view from the top down. In

The Routledge handbook of discourse processes (pp. 7-21). Routledge.

Kintsch, W., & van Dijk, T. A. (1978). Toward a model of text comprehension and production. *Psychological Review*, *85*(5), 363–394. https://doi.org/10.1037/0033-295X.85.5.363

Leeuw, J. R. de, Gilbert, R. A., & Luchterhandt, B. (2023). jsPsych: Enabling an open-source collaborative ecosystem of behavioral experiments. *Journal of Open Source Software*, *8*(85), 5351. https://doi.org/10.21105/joss.05351

Letovsky, S. (1987). Cognitive processes in program comprehension. *Journal of Systems and Software*, *7*(4), 325–339. https://doi.org/10.1016/0164-1212(87)90032-X

McDaniel, M. A., & Butler, A. C. (2011). A contextual framework for understanding when difficulties are desirable. In *Successful remembering and successful forgetting: A festschrift in honor of Robert A. Bjork* (pp. 175–198). Psychology Press.

McNamara, D., & Magliano, J. (2009). Toward a comprehensive model of comprehension. *Psychology of Learning and Motivation - Advances in Research and Theory*, *51*, 297–384. https://doi.org/10.1016/S0079-7421(09)51009-2

McNamara, D. S., & Kintsch, W. (1996). Learning from text: Effects of prior knowledge and text coherence. *Discourse Processes, 22*, 247–287.

O'Reilly, T., & McNamara, D. (2007). Reversing the reverse cohesion effect: Good texts can be better for strategic, high-knowledge readers. *Discourse Processes*, *43*, 121–152. https://doi.org/10.1080/01638530709336895

Rugaber, S. (2000). The use of domain knowledge in program understanding. *Annals of Software Engineering*, *9*(1), 143–192. https://doi.org/10.1023/A:1018976708691

Scalabrino, S., Linares-Vásquez, M., Oliveto, R., & Poshyvanyk, D. (2018). A comprehensive model for code readability: A comprehensive model for code readability. *Journal of Software: Evolution and Process*, *30*(6), e1958. https://doi.org/10.1002/smr.1958

Schulte, C. (2008). Block Model: An educational model of program comprehension as a tool for a scholarly approach to teaching. *Proceedings of the Fourth International Workshop on Computing Education Research*, 149–160. https://doi.org/10.1145/1404520.1404535

van Dijk, T. A., & Kintsch, W. (1983). *Strategies of Discourse Comprehension*. Academic Press.

Wiedenbeck, S. (1986). Organization of programming knowledge of novices and experts. *Journal of the American Society for Information Science*, *37*(5), 294–299. https://doi.org/10.1002/(SICI)1097-4571(198609)37:5<294::AID-ASI3>3.0.CO;2-H

Wiese, E. S., Rafferty, A. N., & Fox, A. (2019). Linking code readability, structure, and comprehension among novices: It's complicated. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, 84–94. https://doi.org/10.1109/ICSE-SEET.2019.00017