In Proceeding of the 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)
Melbourne, Australia, May 15-16, 2023, ACM DL (pages 254-265)
https://doi.org/10.1109/ICPC58990.2023.00040

Too Simple? Notions of Task Complexity used in Maintenance-based Studies of Programming Tools

Patrick Rein, Tom Beckmann, Eva Krebs, Toni Mattis, Robert Hirschfeld
Hasso Platter Institute, University of Potsdam
Potsdam, Germany
Email: firstname.lastname@hpi.uni-potsdam.de

***Abstract*—Researchers conducting studies on programming tools often make use of maintenance tasks. The complexity of these tasks can significantly influence how participants behave. At the same time, the complexity of tasks is difficult to pinpoint due to the many sources of complexity for maintenance tasks. As a result, researchers may struggle to deliberately decide in which regard their tasks should be complex and in which regard they should be simple.**

To help researchers deliberately influence task complexity, we discuss different factors of task complexity. We draw these factors from 23 selected and 39 surveyed studies on programming tools. We arrange the factors according to a task complexity model from ergonomics research that we adapt for maintenance tasks. We illustrate the application of the factors through an example critique of a task design. In the end, task complexity might always be too complex to be fully controlled. Nevertheless, we hope that our discussion helps other researchers to decide in which dimensions their tasks are complex and in which dimensions they want to keep them simple.

***Index Terms*—task complexity, empirical studies, programming tools, software development tools, survey**

I. Introduction

Researchers working on programming tools empirically study programmers through experiments or user studies [1]. A typical setup of such studies revolves around maintenance tasks that "take the form of an addition, removal or debug task carried out on a piece of code" [2]–[4]. In these setups, researchers face the challenge that the complexity of the maintenance tasks can influence the behavior of programmers. In this paper, we aim to provide guidance to researchers on how to analyze and, to some degree, shape the complexity of tasks.

Tasks are a central component of study setups. Correspondingly, their selection and design are prominently discussed in papers that describe strategies to design studies on programming tools [1], [3], [4]. Generally, the tasks are often the main stimulus for participants, next to the tool under investigation. The characteristics of the tasks may influence the behavior of programmers with regard to, for example, their overall comprehension strategy, whether they use concrete or symbolic mental simulation, or whether they debug opportunistically or systematically [5]–[7]. As a result, the tasks may determine the explanatory power of an experiment or the usefulness of observations in user studies and their interpretation [3, p. 112].

Tasks have multiple characteristics that influence programmer behavior. One such characteristic is *task complexity*. Task complexity can have a profound impact on programming and comprehension strategies employed by programmers [5], [6]. For example, programmers may employ different debugging strategies when working on a small method in comparison to working on a large system [7]. The problem with task complexity is that it is composed of several factors. For example, while the complexity of a task often depends on the size of the source code, the complexity might also result from other characteristics such as the kind of defect to be fixed, the quality of the source code, or the presence or absence of additional documentation. Thus, a task can be complex in some regard and simple in others. This variety of characteristics makes it difficult for researchers to consciously and comprehensively decide for which of these characteristics their task should be simple or complex.

With this paper, we want to support researchers investigating programming tools by enabling them to analyze and shape the complexity of maintenance tasks so that those are appropriate for their research questions[1]. Therefore, we describe a collection of factors that contribute to task complexity, derived from 62 publications sourced from an existing corpus on program comprehension studies, additionally selected studies, and our own experience with similar studies. The contribution of this collection is not a complete theory of the complexity of maintenance tasks, but a practical reference for researchers designing studies in the absence of such a theory. To support researchers in finding task complexity factors relevant to their study, we arrange the factors according to task artifacts [8]. To make the collection useful to studies with varied theoretical underpinnings, we avoid prescribing too many theoretical constructs on program maintenance and thus structured the collection along the task artifacts. To further support researchers in selecting task complexity factors, we also associated the factors with task complexity dimensions from a comprehensive and generic framework of task complexity used in ergonomics research [9]. To keep our collection focused, we limited our scope to studies with trained programmers. Within this scope, our collection can be used to analyze purposefully designed tasks as well as tasks retrieved from existing projects.

The more general topic of setting up studies on software development is covered by numerous related guides and stud-

[1]Parts of Sections I and II are based on a publication of preliminary results with the Programming Experience Workshop 2022 (PX/22) [8].

ies [1], [3], [10], [11]. Some of these guides also cover task characteristics to some degree. In contrast to these works, we focus exclusively on the design of tasks and discuss them in detail.

In the following, we define task complexity, contrast it to task difficulty, illustrate the role of maintenance tasks in studies on programming tools, describe our methodology, and briefly introduce the task complexity framework used (Section II and Section III). We then present our collection of task complexity factors (Section IV). To illustrate how researchers may apply our guide, we critique an example task from a study on programming tools (see Section V).

II. Maintenance Task Complexity in Studies

Maintenance tasks are used in a variety of studies on programming tools. In this paper, we focus on perfective and corrective tasks. To lay the foundation for the subsequent analysis, we define task complexity and contrast it to task difficulty. While task complexity is seldom used and defined in programming tools studies, we show examples of studies that already discuss some characteristics that contribute to task complexity. Finally, we briefly point out how researchers in other fields tackle task complexity.

A. Studies based on Maintenance Tasks

Software maintenance tasks are tasks in which participants are presented with an existing program or system and some form of description of a desired change or outcome. Such tasks are used in various kinds of studies, from controlled experiments to observational studies with flexible setups [1]–[3], [12], [13]. There are typically two forms of maintenance tasks: perfective and corrective [14]. In both kinds of maintenance tasks, participants receive a program or system and some description of the desired change. Maintenance tasks are used by researchers evaluating existing or new tools as well as by researchers developing cognitive models or theories of programming activities [13], [15], [16].

In *perfective* maintenance tasks participants are asked to adapt existing features or add new features according to some description of the new behavior [14]. For example, in an evaluation study for an Android programming tool, the task was to add a database import to an existing Android app [17]. An observational study on how programmers gather information on their programs, asked participants to implement five features in a small painting application [13].

In *corrective* maintenance tasks, participants are asked to repair defective behavior [14]. Corrective tasks are often used to evaluate debugging tools and strategies. For example, an experiment using corrective tasks explored whether the live feedback in spreadsheets helps programmers during debugging [12], [18]. Participants received two small, synthesized spreadsheets in two different domains and were asked to repair as many defects as possible within 15 minutes. An experiment evaluating the Whyline tool asked participants to repair two real defects in a large project [19].

B. Defining Task Complexity

In subsequent sections, we will explore how researchers currently deal with the complexity of their tasks and how task complexity can influence programmer behavior. In preparation, we first define the term *task complexity*.

The definition we use defines task complexity as: "the aggregation of any intrinsic task characteristic that influences the performance of a task" [9]. Hence, we regard task complexity as a compound concept that subsumes other characteristics of tasks [9]. The criteria that the characteristics have to influence the task performance are explained as: "If a task characteristic imposes specific resource requirements (e.g., cognitive and physical demands, required knowledge and skills) on task performers, it is considered to influence the performance of the task" [9]. Correspondingly, task complexity describes characteristics of the task that may influence generic performers.

Task difficulty is related to task complexity. However, in contrast to task complexity, task difficulty depends on individual task performers. For this work, we use the definition that task difficulty is the effort task performers perceive when working on a task [9]. Thus, task difficulty results from the combination of task complexity and the personal resources of performers. Examples of such personal resources in the context of programming tools are how experienced task performers are in programming and how much they know about the application domain [20], [21]. Researchers often use task difficulty to describe their tasks [1], [3] and commonly assess or shape task difficulty through expert judgment or piloting.

In summary, task complexity and task difficulty are both relevant to study designs. While task complexity is relevant to the actual research questions, task difficulty is also a practical concern for study designs, as it determines, for example, how much time participants will spend on a task. Matching task complexity and participants is a common challenge, in which analyzing task complexity can help determine sources of complexity that may become difficult for participants.

C. Task Characteristics in Studies using Maintenance Tasks

The tasks used in studies are a major influence on participants. As they determine how useful and generalizable observations are, researchers discuss their tasks in detail via a variety of characteristics. An example of such a description is an evaluation study of a new user interface metaphor for IDEs [22, p. 2509]. The description covered, among other aspects, the overall system size in several metrics such as lines of code and number of classes, the size of the affected features, and the kinds of defects. Further, the researchers also controlled some characteristics, such as the technical knowledge required and the presence of documentation. Similarly, an evaluation study of test-based fault navigation tools describes detailed characteristics of debug tasks [23, p.92]: the length of the infection chain, the presence of tests, and whether the defects are wrong or missing code.

These studies already describe several characteristics that contribute to task complexity. Nevertheless, as there is no structured guidance on task complexity, potentially relevant

characteristics are often missing. An example of a task description that might benefit from a more thorough discussion of the tasks is a user study on an Android prototyping framework [17, p. 102]. This study outlines the application domain of the app to be adapted and a brief description of the feature to be implemented.[2] To fully understand the subsequent observations, readers may also benefit from a description of the size of the original system and an ideal patch implementing the new feature. Readers may also benefit from the description of characteristics that are seemingly unrelated to a perfective maintenance task but still contribute to the task complexity, such as how much of the system participants needed to understand to implement the feature, and how many steps were required to evaluate whether the implemented functionality met the specification.

D. Task Complexity in other Research Areas

Task complexity is a common concept used to characterize tasks in a variety of research areas. A model of task complexity with broad applicability is the model of hierarchical complexity [24]. The model uses information theory to define the complexity of tasks either horizontally [24, p. 250] or vertically [24, p. 251]. While this model is quite generic, it also only describes the information theoretical nature of task complexity, without references to the actual characteristics of tasks. In IR research, task complexity is used to characterize search tasks [25]. Researchers studying IR use task complexity in studies to evaluate search systems or observe usage patterns to inform theories of information retrieval. In industrial ergonomics research, task complexity is used to assess processes, such as continuous monitoring of a nuclear power plant [9], [26], [27]. Researchers use task complexity to analyze the structure of the processes and determine complexity factors that may be reduced.

E. Related Guides on Tasks in Experiment Designs

Numerous guides, models, and literature studies discuss the general topic of setting up studies on software development. A recently published guide on experiments on code comprehension also describes characteristics of the tasks and the code participants work on [3]. Another guide on the general process of evaluating software engineering tools through experiments discusses practical considerations of task sourcing and timing [1]. A study on the state of experimentation in software engineering research describes the spectrum of kinds of tasks that are used in experiments and relates them with the sizes of the tasks [4]. Similarly, a survey of code readability studies outlines the characteristics used throughout studies and also describes the evaluation methods used and how they relate to learning activities [28]. Finally, a study on confounding variables in research on program comprehension finds and discusses several task complexity properties [11].

[2]This observation only refers to the list of task characteristics given in the paper, which might be brief due to a page limitation. The authors may very well have considered other task characteristics when selecting the tasks.

Beyond general guides, there is also a related model that aims to quantify task complexity of maintenance tasks in general, based on a model from psychology [29], [30]. At the fundamental level, the model uses changes to source code and information cues that are required to perform the changes. Based on these primitive elements, the model distinguishes between component complexity (number of changes and the information required for each), coordinate complexity (based on the relations between changes, for example, pre-requisites or timing), and dynamic complexity (changes during the task that influence the relations between changes).

III. Collection Structure and Methodology

Our collection comprises task complexity factors and considerations from 62 papers. To make the collection accessible, we grouped factors according to the variation points of tasks that researchers can influence, such as the task description, the system, or the patch to be created. To support researchers further in shaping the task complexity for their respective research questions, we also grouped the factors according to general dimensions of task complexity.

A. Variation Points for Maintenance Tasks in Studies

To identify sources of complexity in program maintenance tasks, we considered models that describe general activities of maintenance tasks [31]–[33] [34, p. 191]. Speaking in general terms, these models postulate that programmers would begin with an initial comprehension phase, where materials such as a feature description, an observed fault, or similar are analyzed. Next, programmers would move on to identify the code in the software system that is relevant for implementing the feature or repairing the fault. In the case of corrective maintenance, the programmers repeatedly formulate and test hypotheses, for example using their existing knowledge of the software system or entry points derived from the comprehension phase, trying to reproduce and understand the concern. In the case of perfective maintenance tasks, they will work to understand the context of the feature to be implemented and the potential impact changes might have on other system parts [33]. Finally, the programmers create a patch that should eventually yield the improved program.

Drawing from these activities, we extracted distinct variation points that researchers can affect in their study setup: the *task description*, the software *system*, the *infection chain*[3] *or feature location*, the *patch* participants are expected to create, and the *tool environment*.

B. Integrating General Complexity Contributing Factors

We want to support researchers in determining which factors might be relevant to their research question. To support them in fine-grained decisions, we not only classify the collection of factors based on variation points but also based on general sources of task complexity. We draw these general

[3]The infection chain refers to the steps between the instruction (defect) that creates an erroneous run-time state (infection) and the instruction that leads to the wrong surface behavior of the program (failure) [7], [23].

sources from a generic framework of task complexity that is supposed to provide a generic perspective on task complexity independent of particular domains [9]. This generic framework is the result of a review of 24 previous models of task complexity. It provides a set of five components affecting task complexity: goal and output, input, process, time, and presentation. For each component, the framework defines complexity-contributing factors (CCFs), which are concrete factors that increase or decrease the complexity of its component.

The first component describes *goal and output* factors; its CCFs affect the abstract goal of the task through characteristics such as clarity, quantity, or redundancy of the goal. Second, the *input* factors describe the materials and stimuli given to participants, for example, their rate of change, quantity, or conflict. Third, the *process* component includes factors such as the required quantity of actions, repetitiveness, or cognitive requirements of the process. Fourth, the *time* component comprises the factors of concurrency and time pressure of the task. Finally, the authors of the framework chose to consider *presentation* as a separate component, noting that it may also be considered part of the input component.

For each variation point, we classified the factors we identified according to these CCFs. With the CCFs, researchers can take more fine-grained decisions. For example, researchers investigating a tool for keeping track of design decisions may want the task to be complex due to the size of the patch (patch: output quantity) and the trade-offs to be taken during patch creation (patch: output clarity, output conflict), but not because of an ambiguous description of the task description (task description: input clarity), or misleading code comments (system: input conflict).

C. Survey Methodology

The goal of this work is not a comprehensive survey of papers, but a guide covering a wide spectrum of task characteristics. Thus, we did not aim to completely cover all works for a single well-defined area but instead aimed to survey a wide variety of works from different communities.

We started with 23 publications that we were either familiar with, or that we knew were well-received in their respective communities. While this formed a good basis, we tried to reduce the impact of our own selection bias by sampling further factors. Therefore, we sampled the corpus of a related literature study on confounding factors in studies on program comprehension [11], resulting in additional 39 papers. In combination, this results in a corpus of 62 papers.

For surveying the publications from the related literature study and guides, we followed the SALSA (Search, AppraisaL, Synthesis, Analysis) structure for surveys [35].

Search: Our initial corpus is the corpus that resulted from the search phase of the related study on confounding factors in studies on program comprehension [11]. This corpus contains publications on empirical studies of any kind and is not yet filtered by any further criteria. As a result, we started with 842 publications on studies with human subjects published between 2001 and 2010 at publication venues such as ICSE, CHI, and WCRE (for a full list see [11, p. 3]).

Appraisal: During appraisal, we selected publications that covered studies or theories on programming tools in the widest sense. Thus, we also accepted methodologies, such as test-driven development, and language extensions, such as aspect-oriented programming (AOP). We further only selected publications that discussed characteristics of maintenance tasks or described studies that used maintenance scenarios. We included all kinds of maintenance tasks covering perfective, adaptive, and corrective tasks on the system code as well as on other artifacts describing the system (for example architectural diagrams or automated tests). Finally, we excluded publications that included only minimal descriptions of tasks, such as only numerical characteristics of the used system, or only a description of the general application domain.

We also excluded studies that mentioned that they focused on "end-user programmers". As we did not encounter borderline papers, we did not use a specific definition of end-user programmers, but only the term as exclusion criteria. Our findings may apply to such studies, but we were not confident that our corpus would adequately cover the considerations required for studying end-user programmers.

In a first appraisal round, one of the authors rejected any paper on topics other than software development based on the title or if in doubt keywords and abstract, resulting in a set of 177 publications. The number of publications decreased so much, as the initial corpus also contained papers from human-computer interaction publication venues that only occasionally include software development papers. In a second appraisal round, two of the authors selected studies that used maintenance scenarios or investigated some characteristics of maintenance tasks. The two authors worked on two distinct subsets of the corpus but discussed any publications for which the decision may be ambiguous. The result of the second appraisal round was a set of 68 publications.

In a third round, three of the authors went through all 68 publications and read through the studies in detail to determine whether they describe their tasks in sufficient detail. All three authors first read and judged the publications individually before discussing all candidate publications and agreeing on the final decision to include the publication. As a result, the appraisal phase yielded a corpus of 43 publications, including 4 papers also in the initial 23 papers.

Synthesis: To gather as many factors from the selected papers, three of the authors read through the selected publications and extracted any factors of task complexity that they found notable. The authors did not distinguish between whether experimenters were aware of factors or not and extracted factors that were explicit parts of the experiment design or only reported with little theoretical rationale. They also individually assigned them to a variation point and corresponding task complexity component (see Tables I and II). Again, each of the authors read all publications on their own. After completing the individual synthesis, the three authors discussed all publications and for each factor they had

found, they decided together on the variation point and task complexity component.

Analysis: We did not analyze the resulting collection of factors beyond the descriptive discussion of specific combinations of variation points and CCFs. We did so for two reasons.

First, our collection is designed to be purely descriptive and we aimed to impose only enough theory on it to make it accessible. By analyzing it further, we would impose more theoretical assumptions onto the collection, thus making it potentially applicable to fewer research questions.

Second, as we surveyed for variety and not for completeness, the resulting collection of factors does not allow for definitive statements about the state of task design. We deliberately refrained from any quantitative analysis, as most of the factors we observed were used with little theoretical backing. By quantifying how often which factor was used, we would suggest a consensus between experimenters and that some factors were actually important, which is not clear at this point.

IV. Collection of Task Complexity Factors

In the following, we describe task complexity factors for tasks in program maintenance studies. Tables I and II provide the complete list of factors we have identified in the collected publications (see Section III). The following discussions of the variation points generally arrange the factors of each variation point as listed in the table and elaborate on factors that pose interesting trade-offs or have noteworthy considerations (marked with $^{+}$ in Tables I and II).

Through our collection, we want to help researchers in reasoning about or shaping the complexity of their maintenance study tasks. While we list factors that can influence complexity, researchers will still have to align these factors with their research questions. Also, our collection does not provide guidance on other considerations such as external validity, task difficulty, learning effects between tasks, or the duration of tasks. They are covered in much more detail in other guides [1], [3], [11]. Finally, our collection is concerned with the complexity of maintenance tasks used in studies and not in software maintenance in general [36].

A. Task Description

The initial stimulus for participants to engage with the task will most likely come from a task description. These may correspond roughly to the initial prompt programmers would receive to begin the comprehension phase, such as a bug report or a feature specification. Relevant factors affecting complexity include the task's clarity and ambiguity, but also the format of its presentation and the amount of guidance.

(Output Quantity) Number of Sub-tasks: Having multiple sub-tasks at once requires participants to decide in which order they work on them and requires participants to distinguish between information relevant to the different sub-tasks [37, p. 891], [38, p. 4], [39, p. 8].

(Input Quantity) Size of Task Description: The size of the task description is a basic factor that may influence the complexity of the overall task. Independent of the content and its clarity, a larger document requires participants to read through, assess, and remember more information. However, researchers may not always have full control over the size of the task description. Synthesized tasks give researchers full control of the size of the task description [37, p. 891] [13]. Contrarily, researchers have less control over the size of task descriptions collected from existing projects [40], as they may include details irrelevant to the study, such as other business use cases that are irrelevant to the isolated situation of the study. Similarly, if the study is merely observing programmers working on their usual projects [41], an assessment of the size of the task description may only be possible after the study.

Input Guidance: If the task description accidentally hints at the class containing the root cause of a failure, participants might search for a defect less extensively, as they might already find it using this unintentional hint. In contrast, some guidance can also make it more feasible to use a large system or a large set of sub-tasks in a study. For example, one study provided pointers to two relevant classes out of the 301 classes in the system [37].

B. System

Both in the comprehension and in the defect location phase, the system in which the defect or missing feature is located plays a major role. Participants need to understand it well enough to form hypotheses about where defects may be located or where a feature might be implemented. Thus, the qualities of the system directly influence the complexity of the task, for example through its size, the clarity of its code, or redundancy in the means for participants to explore the systems.

Input Clarity: With regard to clarity, we identified three dimensions in the surveyed work.

First, the complexity of the underlying control flow influences how well the code is understood. The control flow complexity is either characterized by a simple property such as cyclomatic complexity or by some more involved measures that take the cognitive complexity of program elements into account [42]–[44].

Second, beyond this basic complexity, the overall quality of the code also influences how well the system and task is understood. "Spaghetti code" [45, p. 33] and bad code [31, p. 21] have both been brought up by developers when asked about reasons for difficult defects.

Third, the clarity of the system also results from the architecture and the resulting modularity of the code base. For example, in an architecture that directly maps concepts of the application domain to classes, participants may have to infer less to find code relevant for some application behavior [46]. Beyond these three dimensions, researchers might discover further relevant characteristics through the cognitive dimensions of notations framework [47].

All of these dimensions are related to the concept of source code readability [48], [49]. Thus, theory and study results on general source code readability may also inform the choice or design of the system [28]. For example, a model of the evolution of source code readability may help determine a suitable time in the history of a system [50]. Studies on code readability for specific kinds of source code may help when designing specialized tasks, for example, factors influencing the readability of tests [51].

While all these general qualities of code are difficult to measure, they can still be judged and described in general terms by researchers looking into systems that may serve as the foundation for their studies. Alternatively, code readability analysis tools may help estimate the clarity [49], [52], [53].

System Domains: While the system's domains and the participant's required knowledge are not represented by the task complexity framework, we still include them, as it is often reported by studies [17], [22], [33], [54], [55].[4] This includes knowledge about the application domain [54], [55], [57], as well as knowledge about the technical mechanisms used in the system [22], [58], [59]. Programmers use different strategies for understanding a system depending on whether they have prior knowledge about the application domain [60]. Researchers can control to which extent the application domain is a source of complexity by using an application domain that either a lot of programmers or very few programmers have previously worked with. Technical knowledge such as working with database interfaces or knowing special language features also increases complexity [3, p. 108]. One study explicitly wrapped all such APIs to prevent this as a source of complexity [22, p. 2509]. Another study ensured that one group of participants knew the architectural patterns, whereas another group did not [44].

C. Infection Chain and Feature Location

With regard to the defect or feature location phase, researchers have to decide on the nature of the infection chain or how the feature to be adapted is distributed in the system. In corrective maintenance tasks, participants determine the code section that includes the defect that should be repaired. For example, a failure that no longer occurs upon observation [45, p. 33] is likely to introduce significant complexity. When performing perfective maintenance, before modifying the code, participants first need to determine the sections that implement the feature or the general behavior. For example, a feature that requires changes in multiple packages is likely to be more complex than one that only requires a change to a single package.

(Output Clarity) Type of Failure: Different kinds of failures communicate different amounts of information that can be used to track down the defect. For instance, a crash due to an exception provides a starting point to determine the erroneous state and follow the infection chain backward [7]. In contrast, wrong behavior may not provide such a clear starting point and require participants to first relate the observed behavior to the run-time state [9], [22].

(Output Clarity) Type of Defect: Defects can be classified and described along a variety of dimensions, all having different impacts on task complexity [83], [93].

A major distinction is the one between defects of commission and defects of omission [92] [3, p. 111] (commission subsumes mechanical and logic errors, categories that are also often used [93]). A defect of commission is manifested as a wrong piece of code. Participants can spot it while reading the code. For example, in one study, the complexity of actually spotting the root cause was deliberately reduced by commenting out important code [23]. In contrast, defects of omission result from missing code. This is more complex to determine, as participants have to understand the code well enough, to realize that a statement is missing.

D. Patch

Once participants understood the defect or located the places to add the requested feature, they enter the phase of creating a patch to address the concern. The patch is formed through the set of modifications the participants propose to solve the task. Following the creation, participants also have to evaluate whether the patch is appropriate. Complexity introduced through the patch is determined through factors such as the patch's size, its degree of scattering throughout the code base, or the degree of novelty as opposed to solving by copy-paste.

(Output Quantity) Size of Patch: The larger the patch needs to be, the more decisions participants need to take in order to create it. Some studies try to prevent that the patch generation adds any complexity by making the patch a minimal edit such as uncommenting a statement [23] or hinting at the specific type of change required, such as a variable renaming [63]. Other studies keep track of the size of the minimal or typical patch [22] [37, p. 891].

(Output Conflict) Conflicting Requirements: When creating the patch, participants might wonder whether a patch only needs to meet the functional requirements or also needs to fit into the existing architecture and match the coding style. Similarly, for corrective tasks, participants may wonder whether they should simply repair the surface behavior or aim to repair the root cause of the failure. While either one may be acceptable for the research question, participants may struggle when there are no clear instructions on what counts as *repairing* the failure. Further, the original requirements for the patch might be conflicting. One study explicitly introduced complexity in that regard by asking participants to "make the design as ideal as possible by the criteria of performance, understandability, and reusability" [33, p. 363].

E. Tool Environment

Not explicitly stated as part of the model on debugging phases, we argue that researchers should also consider the tool environment in which their study is embedded. If the study aims to evaluate a tool, this tool's complexity is inherent to the

[4] Knowledge is considered a contributor to task complexity by other models [56].

TABLE I
OVERVIEW OF COMPLEXITY-CONTRIBUTING FACTORS AFFECTING THE TASK VARIATION POINTS. ABBREVIATIONS REFER TO TASK COMPONENTS: OUT(PUT), INP(UT), PRO(CESS), PRES(ENTATION).

Var. Point	CCF	Interpretation for Software Maintenance Tasks
Task Description (IV-A)	Out. Quantity$^+$	one task [19], [23] or multiple tasks at a time [37, p. 891], [38, p. 4], [39, p. 8]
	Out. Redundancy	ensuring that the same patterns do not occur in different tasks [54]
	Inp. Quantity	length of task description [9] (e.g. large specification documents [61, p. 509])
	Inp. Clarity	explicitly stated target behavior (e.g. from a standard) or derived behavior (e.g. through demonstration of the tool [62, p. 7] [63], [64])
	Inp. Inaccuracy	allowing arbitrary feature additions [65, p. 249]
	Inp. Guidance$^+$	deliberate hints to the defect location / feature location [37, p. 891], [66, p. 5], [63, p. 5], avoiding hints [67, p. 5], [68, p. 5], [69], explicit sub-tasks that lead through the process of locating defect / feature and generating the patch [68], [70], ordering of tasks guides through system [57], [71]
	Inp. Mismatch	task description in different natural language than other documents or identifiers in code [72], [73]
	Inp. Redundancy	multiple ways to understand the task (e.g. natural language description and tests [23], [37], [74], screenshot of result [63], demonstration of result [62], [63])
	Pres. Format	using corresponding diagrams (state diagram for target automaton [75, p. 604], sequence diagrams of main scenarios to be implemented [73, p. 69])
System (IV-B)	Inp. Quantity	LOC, NOM, NOC, NOP, etc. [13], [22], [23], [40], [41], [44], [54], [58], [59], [62], [74], [76]–[81], quantitative characterization of the system part that is relevant to task [13], [22], [63], [76], [79], feature set [66], [68], [74]
	Inp. Clarity$^+$	domain: complexity and quality of algorithms [5], [43], [54], [31, p. 21] control flow: CYCLO, cognitive complexity [42]–[44] architecture: closeness of mapping from domain to code [46], suitability of architecture for the patch [73, p. 67], coupling and cohesion [82, p. 988], usage of specific abstractions [83] other: system is based on multiple languages [77], system only uses a reduced version of a common language [84], syntactic characteristics [83]
	Inp. Mismatch	surprising use of programming language [3, p. 108 f.], confusing identifiers [3, p. 108 f.], speaking identifiers [73, p. 69], deliberately obfuscated identifiers [67, p. 5]
	Inp. Inaccuracy	complete source code available or only sections [22] (e.g. only client code without server code [85, p. 5])
	Inp. Redundancy	providing no additional documentation [13], [22], removing existing documentation from code [86, p. 3], providing additional material describing the behavior (user manual [37, p. 891], [40], [62], [78], [87], specification [39], object diagram / class diagram [87, p. 5]), providing entrypoints to code [79]
	Inp. Conflict	documentation contradicts source code ((deliberately) obscuring comments [37, p. 891])
	Inp. Guidance	system walkthrough pre-experiment and explanation of major parts of the system [63]
Infection Chain / Feature Location (IV-C)	Out. Quantity	number of source code locations [39], [88], [89] (reduce number of relevant modules [90], mark regions that are potential sources [63]), length of infection chain [31], [45], number of failures per task [13], [23]
	Out. Clarity$^+$	type of failure (compilation error, crash, wrong behavior, missing behavior [7], [22], [39]), type of defect [22], [23], [76], [83], [91] (e.g. null handling, accidental modification of meta-objects), commission or omission [92], [93], [3, p. 111] (e.g. commenting out code instead of removing it [23]), defect can be in system or test code [57], declaring number of defects [22], [23] or only existence of defects [61, p. 509], [57, p. 3]

study. However, programming tools are rarely used in isolation and the choice or availability of additional tools can have an impact on the complexity of the task. For example, tools that aid participants in navigating and analyzing the software system, such as a means to browse references, can support formulating hypotheses, while a debugger or a REPL can help in testing hypotheses.

F. Overall Considerations

Finally, some factors may influence task complexity beyond the individual variation points and throughout the overall study design. These factors can include imposing and communicating a time limit [73, p. 69] [19], providing guidance or interventions from the researchers [33], or non-routine events that interrupt the participants [13], [63], [80].

V. EXAMPLE CRITIQUE OF A MAINTENANCE TASK

Our collection describes a list of factors that contribute to task complexity. However, as this is a general list, researchers will still need to align the factors with the research questions of their studies. To illustrate how our collection may be used in this regard, we critique a task from a preliminary study of ours with regard to the complexity-contributing factors. The full analysis of all factors is beyond the scope of this example analysis, thus, we focus only on interesting factors relevant to the study. For each factor, we describe whether we aim to make it simple or complex and why. For some factors, the complexity is a result of other decisions and we can not influence it. In these cases, we analyze the expected complexity and whether this aligns with the research question.

The task is part of an unpublished study on the influence

TABLE II
CONTINUED OVERVIEW OF COMPLEXITY-CONTRIBUTING FACTORS. ABBREVIATIONS REFER TO TASK COMPONENTS: OUT(PUT), INP(UT), PRO(CESS), PRES(ENTATION).

Var. Point	CCF	Interpretation for Software Maintenance Tasks
Patch (IV-D)	Out. Quantity$^+$	size of patch (size of minimal / typical patch [37, p. 891] [22], [44], [84], measuring size of patches during experiment [77], [94]), size of element to be changed (change of number / change to method [71, p. 5]), specifying type of change (rename variable [63]), minimizing patch size via a statement that is commented out [23]
	Out. Clarity	matching changes to locations (changes across files [37, p. 891], [40], [44], system structure prepared for the patch [64, p. 5] [54, p. 169], specifying a section where changes are to be made [63]), providing a skeleton to guide implementation [58], [95]
	Out. Conflict$^+$	writing the patch in actual code or as change recommendation [96] or as general natural language description [81], root cause vs symptoms, asking to optimize several dimensions at once (e.g. performance, understandability, and reusability [33, p. 363]), making quality expectations explicit (e.g. prototyping not production-level code [97])
	Out. Redundancy	multiple valid solution and no clear metric to decide (explicitly allowing multiple solutions [13], patch designed as minimal edit [23], [59]), patch already exists in similar form in system [54], providing sample code to account for unfamiliarity with API [63], [78], [94]
Tool Environment (IV-E)	Pro. Clarity	providing tools to reproduce behavior (test runner [23], [37, p. 891], [59], tests and mock objects [75, p. 604]), explicitly prescribing the process (e.g. TDD [70, p. 6])
	Pro. Qty of Paths	availability of different tools (noting available standard tools [22, p. 2510], [34], [63], disabling other tools to various extents [19], [76], [79]), optional tool for control group that corresponds to tool under test [63], tasks can be accomplished either with tool under test or with built-in IDE features [59], [63]
	Pro. Conflict	avoiding tools to mitigate impact of technical problems (using pen and paper [84], writing change proposal instead of code [96]), tools masking defects or suggesting wrong hypothesis (stepping debugger for a race condition), tool to be used is deliberately not helpful for the task [69, p. 6]
	Pro. Qty of Steps	number of steps to get some particular information (compiling vs REPL workflow or debugger vs printf)
Overall (IV-F)	Time	communicating a time limit [12], [18], solving as many tasks as participants can in a time frame [12], [80], announce reward for fastest and/or most accurate solution [63], [97], time sink task: it does not count and takes long but participants do not know that and thus it creates time pressure [73, p. 69], explicitly state that progress is not relevant [40, p. 437], explicitly ask to emphasize speed over correctness [19]
	Guidance	intervening during experiment to clarify or repair issues [33]
	Non-routine Event	crashes of the tooling, deliberate interruptions [13], [63], [80]

of task complexity on the efficiency of live programming tools used in debugging (for details, see the accompanying artifact [99]). We operationalize debugging efficiency as the time to repair a given failure. Participants either worked with a set of live dynamic tools[5] (live object inspector, live code editing, live UI inspector, edit-and-continue debugger) or a set of baseline programming tools (basic stepwise debugger, basic object inspection). The study is conducted in the exploratory-style live programming environment Squeak/Smalltalk [98].

Programmers work on several tasks, one task after another. We expect dynamic tools to mostly help programmers with localizing defects and evaluating their patches. We do not expect the tools to help participants with understanding the system in general and generating the patch. Thus, we want participants to spend most of their effort on defect localization and patch evaluation. Further, as we are looking into the influence of task complexity, we require tasks that are either simple or complex in relation to each other.

Fig. 1. A screenshot of the game Jump-O-Drom used as the system in our example study

A. System

The selected system determines the way we can influence subsequent variation points. We wanted to investigate the usage of dynamic tools in a scenario beyond a small module, but needed to stay within realistic time limits. We chose a small game as the system, as games combine a variety of concerns such as event handling, state propagation, file I/O, rendering, and algorithms. This variety allows us to define tasks that cover different parts of the system. The game we

[5]Dynamic tools, in contrast to static tools, are tools that work with dynamic information on the system behavior.

1) Open the game settings in the main menu
2) Change the setting “Minigame Selection” to ”JodHotPotatoMinigame”, by using the keys for “left” and “right”
3) Start the game
4) Wait until you see a player explode

Both players explode in quick succession. Only one player should explode and then the potato should go to a random player. The goal of the game mode is that you don’t have the potato when it explodes. The potato can be given to the other player by jumping on them from above.

Fig. 2. An English translation of the task description.

used in our study is named “Jump-O-Drom”. It is a multiplayer jump-and-run game in which most rules can be changed resulting in a large number of different game modes (see Figure 1).

Regarding *quantity (input)*, the game has a large feature set: configurable game modes, configurable physics, configurable player appearance and controls, extensible collision handling, level editor, temporary effects on players, abilities for players, sound, and custom widgets and menu classes. At the same time, the source code can be considered small (3052 LOC, 73 classes, 759 methods, 8 packages).

We were mostly interested in how programmers apply dynamic tools to fix a defect, not how they use them to learn about a system in general. Thus, we provided *guidance (input)* with regard to the general system behavior by introducing the module structure and important classes of the game. We used a fixed script to avoid providing any additional guidance that might influence the complexity of the individual tasks.

Similarly, as we were not interested in how programmers explore the present behavior of a system, we made the description of the game behavior *redundant (input)* by giving an interactive tutorial on the gameplay.

Concerning *clarity (input)*, we wanted to ensure that we do not observe tool usage resulting from unnecessarily complex code or a convoluted architecture. Thus, we ensured that the project contains no significant idiomatic or architectural flaws.

B. Task Description

We were not interested in how or how well participants can comprehend the description of the failure. Thus, we aimed to reduce the complexity in the task description (see Figure 2).

For one, we aimed at reducing the complexity by keeping the *quantity (input)* down with a concise description of the task (see Figure 2). We included the steps to reproduce the failure, a description of the observable symptoms, and a description of the expected behavior. Further, we aimed to keep the description *clear (input)*. Therefore, we used a consistent structure throughout all tasks, which distinguishes between the steps to reproduce the failure and the observable as well as the desired behavior. The structure was visually reinforced through a dedicated graphical tool presenting the tasks. Further, we used consistent vocabulary for interactions, parts of the game, and observable behavior throughout all tasks.

We want participants to make use of the tools to generate and test hypotheses about the failure. Thus, while the task description should be clear, it should at the same time only provide little *guidance* about the actual process of repairing the failure. Therefore, we aimed to give as few hints on the source code location of the defect as possible. For example, to not give away the class in which the defect is located, we did not use any terms related to the class name.

C. Infection Chain

Our main interest in this study was the way participants use dynamic tools to determine the defect location. Thus, we needed to make the defects complex enough for participants to spend considerable effort on locating them, while at the same time keeping them doable in the available time.[6]

The defect in our example task is a missing reset of the timer that triggers the explosion of a player. As a result, the explosions are directly triggered in each successive game step.

We used the *size (output)* of the infection chain to ensure that the defects could be found in the available time frame. Therefore, we spread the defect, infection propagation, and failure among few classes. In the case of our example class, the whole infection chain is limited to three classes. Also, we did not want to observe special debugging techniques, thus we limited the tasks to one defect per failure.

At the same time, we needed to ensure that participants had to invest enough effort into locating the defect so that they had a reason to use dynamic tools. To achieve that, we used characteristics that influence the *clarity (output)* of the infection chain. For example, the failure should require participants to determine code locations responsible for the observable game behavior first. Thus, our example task and all other tasks in the study are observable as wrong game behavior, and not exceptions or crashes, as they would provide an obvious starting point for tracing the infection chain. At the same time, we did not want participants to make too many assumptions about the intended behavior. Thus, while the defect leads to wrong behavior, the defect is a mere programming error and not a specification error.

Finally, to distinguish between different levels of complexity, we used defects of commission for simple tasks, and defects of omission for complex tasks. Thus, as the missing reset of the timer is a defect of omission, we consider our example task a complex task.

D. Patch

For our study goal, the complexity of the patch should ideally trigger the usage of dynamic tools to determine suitable source locations, inspect objects, and evaluate potential solutions. At the same time, to get comparable observations,

[6]We also considered aspects such as learning effects between defects, but as these are not covered by our collection and other guides on experimentation cover them in much greater detail, we do not discuss them in this paper.

we wanted to keep determining the target behavior simple. So, determining what to implement should be simple, while determining how to implement it should be complex.

In our example task, the fact that the timer needs to be reset should be obvious when the defect is identified correctly. Participants then still have to determine implementation details such as the correct location and methods to reset the timer.

The *clarity (output)* of the location and code for the patch differs between defects of omission and commission. For a defect of commission, determining the location of the patch is simple and the surrounding code already limits the number of possible methods. For a defect of omission, finding a suitable location is more complex and as the missing statement needs to be created, there are fewer constraints on the code to be written.

Determining what to implement is kept simple due to the limited number of *redundant (output)* target behaviors. Due to the small size of the defects, participants should be able to describe the target behavior given that they correctly identified the defect. To make deciding on a target behavior even simpler, we aimed to prevent *conflicting goals (output)*, by explicitly asking participants to work on the patch until they are as sure of it as they would when committing it to one of their own projects.

E. Tool Environment

In our study, we do not study a particular tool, but the general usage of dynamic tools for debugging tasks. As we were interested in the usage frequencies for particular tasks in comparison with other tools, participants had access to all tools available in the Squeak/Smalltalk programming environment.

Regarding the *clarity (process)* of using the tools to observe the behavior, we did not provide automatic tests, as they would be obvious starting points. Further, concerning the *quantity of paths (process)* to employ the tools, we provided no hints on what they should use. To make sure participants are aware of all potential tools, we briefly recap the available tools at the beginning of a run.[7] Finally, to prevent that a larger *quantity of steps (paths)* to get to relevant information prevents the usage of tools, we also recapped keyboard shortcuts and context menus.

F. Discussion

Our collection of task complexity factors guided us through the initial design of our tasks. Through the collection of factors, we were able to get a grasp of the task characteristics and in which regard they may be complex. This in turn allowed us to shape the tasks more deliberately, such as making the infection chain complex instead of the task description or choosing a system with a simple architecture to reduce the effort spend on system comprehension.

In addition to employing the collection, we also ran a pilot study with four prospective participants. For our example task, we observed that most pilot participants found the state error and the defect in code too quickly for interesting strategies to emerge. By putting our observations into terms of our collection of task complexity factors, we determined that the original task description provided too much guidance and that the length of the infection chain was too short. We removed terms related to the defect location from the task description (resulting in the description in Figure 2) and introduced two new methods in the infection chain. In subsequent pilot runs, participants had to put in more effort to reach the infection and find the defect.

[7] We only recap the tools, as participants are familiar with the environment.

VI. Conclusion

We presented a collection of task complexity factors tailored toward software maintenance tasks used in studies on programming tools. It is intended to help researchers struggling to design or choose appropriate maintenance tasks for their studies. By going through the different factors for each of the five variation points of the tasks, we hope that they approach a complete picture of the ways their tasks are simple or complex. Accordingly, they may discover aspects that should be simpler or more complex depending on whether they are relevant to their research questions.

By design, our collection does not provide means to judge task difficulty, as this ultimately depends on the combination of task complexity and the personal resources of the individual participants. The challenge for researchers remains to match task complexity and participants in a way that the task difficulty is suitable for the research questions.

Data Availability

The bibliographies, the intermediary tables of the synthesis, and the study materials are available on Zenodo [99].

Acknowledgments*

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 449591262. We also gratefully acknowledge the financial support of HPI's Research School[8].

[8] https://hpi.de/en/research/research-school.html

References

[1] A. Ko, T. LaToza, and M. Burnett, "A practical guide to controlled experiments of software engineering tools with human participants," *Empirical Software Engineering*, vol. 20, no. 1, pp. 110–141, 2015.

[2] A. Dunsmore and M. Roper, "A comparative evaluation of program comprehension measures," Department of Computer Science, University of Strathclyde, Tech. Rep. EFoCS 35-2000, 2000.

[3] D. G. Feitelson, "Considerations and pitfalls in controlled experiments on code comprehension," in *Proceedings of ICPC 2021*. IEEE, 2021, pp. 106–117.

[4] D. I. K. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N. Liborg, and A. C. Rekdal, "A survey of controlled experiments in software engineering," *IEEE Trans. Software Eng.*, vol. 31, no. 9, pp. 733–753, 2005.

[5] A. von Mayrhauser and A. M. Vans, "Comprehension processes during large scale maintenance," in *Proceedings of ICSE 1994*. IEEE Computer Society / ACM Press, 1994, pp. 39–48.

[6] F. Détienne and E. Soloway, "An empirically-derived control structure for the process of program understanding," *International Journal of Man-machine Studies*, vol. 33, no. 3, pp. 323–342, 1990.

[7] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Elsevier, 2009.

[8] P. Rein, T. Beckmann, T. Mattis, and R. Hirschfeld, "Toward understanding task complexity in maintenance-based studies of programming tools," in *Proceedings of the PX Workshop 2022*. ACM, 2022, pp. 38–45.

[9] P. Liu and Z. Li, "Task complexity: A review and conceptualization framework," *Int J Ind Ergon*, vol. 42, no. 6, pp. 553–568, 2012.

[10] T. August and K. Reinecke, "Pay attention, please: Formal language improves attention in volunteer and paid online experiments," in *Proceedings of CHI 2019*. ACM, 2019, p. 248.

[11] J. Siegmund and J. Schumann, "Confounding parameters on program comprehension: a literature survey," *Empir. Softw. Eng.*, vol. 20, no. 4, pp. 1159–1192, 2015.

[12] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook, "Does continuous visual feedback aid debugging in direct-manipulation programming systems?" in *Proceedings of CHI 1997*. New York, NY, USA: ACM, 1997, pp. 258–265.

[13] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Softw. Eng.*, no. 12, pp. 971–987, 2006.

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

[15] D. A. Boehm-Davis, J. E. Fix, and B. H. Philips, "Techniques for exploring program comprehension," in *Workshop on Empirical Studies on Programmers 1996*, 1996.

[16] X. Rong, S. Yan, S. Oney, M. Dontcheva, and E. Adar, "Codemend: Assisting interactive programming with bimodal embedding," in *Proceedings of UIST 2016*. ACM, 2016, pp. 247–258.

[17] D. Kim, S. Park, J. Ko, S. Y. Ko, and S. Lee, "X-droid: A quick and easy android prototyping framework with a single-app illusion," in *Proceedings of UIST 2019*. ACM, 2019, pp. 95–108.

[18] C. Cook, M. Burnett, and D. Boom, "A bug's eye view of immediate visual feedback in direct-manipulation programming systems," in *Proceedings of ESP 1997*, ser. ESP '97. New York, NY, USA: ACM, 1997, pp. 20–41.

[19] A. J. Ko and B. A. Myers, "Finding causes of program output with the java whyline," in *Proceedings of CHI 2009*. ACM, 2009, pp. 1569–1578.

[20] L. Gugerty and G. Olson, "Debugging by skilled and novice programmers," in *Proceedings of CHI 1986*, ser. CHI '86. New York, NY, USA: ACM, 1986, pp. 171–174.

[21] A. von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[22] A. Bragdon, R. C. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. L. Jr., "Code bubbles: a working set-based interface for code understanding and maintenance," in *Proceedings of CHI 2010*. ACM, 2010, pp. 2503–2512.

[23] M. Perscheid, "Test-driven fault navigation for debugging reproducible failures," Ph.D. dissertation, University of Potsdam, 2013.

[24] M. L. Commons, E. J. Trudeau, S. A. Stein, F. A. Richards, and S. R. Krause, "Hierarchical complexity of tasks shows the existence of developmental stages," *Developmental Review*, vol. 18, no. 3, pp. 237–278, sep 1998.

[25] B. M. Wildemuth, L. Freund, and E. G. Toms, "Untangling search task complexity and difficulty in the context of interactive information retrieval studies," *J. Documentation*, vol. 70, no. 6, pp. 1118–1140, 2014.

[26] P. Liu and Z. Li, "Comparison between conventional and digital nuclear power plant main control rooms: A task complexity perspective, part ii: Detailed results and analysis," *International Journal of Industrial Ergonomics*, vol. 51, pp. 10–20, 2016.

[27] ——, "Comparison between conventional and digital nuclear power plant main control rooms: A task complexity perspective, part i: Overall results and analysis," *International Journal of Industrial Ergonomics*, vol. 51, pp. 2–9, 2016.

[28] D. Oliveira, R. Bruno, F. Madeiral, and F. Castor, "Evaluating code readability and legibility: An examination of human-centric studies," in *Proceedings of ICSME 2020*. IEEE, 2020, pp. 348–359.

[29] L. He, "Software maintainability measurement: A task complexity perspective," Ph.D. dissertation, Mississippi State University, 2010.

[30] L. He and J. C. Carver, "Modifiability measurement from a task complexity perspective: A feasibility study," in *Proceedings of ESEM*. IEEE Computer Society, 2009, pp. 430–434.

[31] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld, "Studying the advancement in debugging practice of professional software developers," *Springer Software Quality Journal*, vol. 25, no. 1, pp. 83–110, 2017.

[32] D. J. Gilmore, "Models of debugging," *Acta Psychologica*, vol. 78, no. 1, pp. 151–172, 1991.

[33] T. LaToza, D. Garlan, J. Herbsleb, and B. Myers, "Program comprehension as fact finding," in *Proceedings of ESEC-FSE 2007*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 361–370.

[34] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *Proceedings of ICSE 2010*. ACM, 2010, pp. 185–194.

[35] M. Grant and A. Booth, "A typology of reviews: An analysis of 14 review types and associated methodologies," *Health Information & Libraries Journal*, vol. 26, no. 2, pp. 91–108, 2009.

[36] R. Leano, S. Chattopadhyay, and A. Sarma, "What makes a task difficult? an empirical study of perceptions of task difficulty," in *Proceedings of VL/HCC 2017*. IEEE Computer Society, 2017, pp. 67–71.

[37] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: An exploratory study," *IEEE Trans. Software Eng.*, vol. 30, no. 12, pp. 889–903, 2004.

[38] L. A. Wilson, Y. Senin, Y. Wang, and V. Rajlich, "Empirical study of phased model of software change," *CoRR*, vol. abs/1904.05842, 2019.

[39] H. Liu and H. B. K. Tan, "An approach to aid the understanding and maintenance of input validation," in *Proceedings of ICSM 2006*. IEEE Computer Society, 2006, pp. 370–379.

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

[41] T. Kelly and J. Buckley, "A context-aware analysis scheme for bloom's taxonomy," in *Proceedings of ICPC 2006*. IEEE Computer Society, 2006, pp. 275–284.

[42] R. Duran, J. Sorva, and S. Leite, "Towards an analysis of program complexity from a cognitive perspective," in *Proceedings of ICER 2018*. ACM, 2018, pp. 21–30.

[43] G. A. Campbell, "Cognitive complexity," in *Proceedings of TechDebt 2018*. ACM, may 2018.

[44] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L. G. Votta, "A controlled experiment in maintenance comparing design patterns to simpler solutions," *IEEE Trans. Software Eng.*, vol. 27, no. 12, pp. 1134–1144, 2001.

[45] M. Eisenstadt, "My hairiest bug war stories," *Communications of the ACM*, vol. 40, no. 4, pp. 30–37, 1997.

[46] B. Meyer, *Object-oriented software construction*. Prentice hall Englewood Cliffs, 1997, vol. 2.

[47] T. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *J Vis Lang Comput*, vol. 7, no. 2, pp. 131–174, 1996.

[48] J. L. Elshoff and M. Marcotty, "Improving computer program readability to aid modification," *Commun. ACM*, vol. 25, no. 8, pp. 512–521, 1982.

[49] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on software engineering*, vol. 36, no. 4, pp. 546–558, 2009.

[50] V. Piantadosi, F. Fierro, S. Scalabrino, A. Serebrenik, and R. Oliveto, "How does code readability change during software evolution?" *Empir. Softw. Eng.*, vol. 25, no. 6, pp. 5374–5412, 2020.

[51] D. Winkler, P. Urbanke, and R. Ramler, "What do we know about readability of test code? - A systematic mapping study," in *Proceedings of SANER 2022*. IEEE, 2022, pp. 1167–1174.

[52] S. Scalabrino, M. L. Vásquez, D. Poshyvanyk, and R. Oliveto, "Improving code readability models with textual features," in *Proceedings of ICPC 2016*. IEEE Computer Society, 2016, pp. 1–10.

[53] Q. Mi, Y. Hao, L. Ou, and W. Ma, "Towards using visual, semantic and structural features to improve code readability classification," *J. Syst. Softw.*, vol. 193, p. 111454, 2022.

[54] M. Vokác, W. F. Tichy, D. I. K. Sjøberg, E. Arisholm, and M. Aldrin, "A controlled experiment comparing the maintainability of programs designed with and without design patterns-a replication in a real programming environment," *Empir. Softw. Eng.*, vol. 9, no. 3, pp. 149–195, 2004.

[55] J. Lawrance, R. K. E. Bellamy, and M. M. Burnett, "Scents in programs: Does information foraging theory apply to program maintenance?" in

Proceedings of VL/HCC 2007. IEEE Computer Society, 2007, pp. 15–22.

[56] P. Ø. Braarud and B. Kirwan, "Task complexity: What challenges the crew and how do they cope," in *Simulator-based Human Factors Studies Across 25 Years*. London: Springer London, 2011, pp. 233–251.

[57] J. Lawrance, S. Clarke, M. M. Burnett, and G. Rothermel, "How well do professional developers test with code coverage visualizations? an empirical study," in *Proceedings of VL/HCC 2005*. IEEE Computer Society, 2005, pp. 53–60.

[58] A. Guzzi, M. Pinzger, and A. van Deursen, "Combining micro-blogging and IDE interactions to support developers in their quests," in *Proceedings of WCRE 2010*. IEEE Computer Society, 2010, pp. 1–5.

[59] S. Hanenberg, S. Kleinschmager, and M. Josupeit-Walter, "Does aspect-oriented programming increase the development speed for crosscutting code? an empirical study," in *Proceedings of ESEM 2009*. IEEE Computer Society, 2009, pp. 156–167.

[60] F. Détienne, *Software design cognitive aspects*, ser. Practitioner series. Springer, 2001.

[61] J. H. Hayes and J. Offutt, "Input validation analysis and testing," *Empir. Softw. Eng.*, vol. 11, no. 4, pp. 493–522, 2006.

[62] T. H. Ng, S. C. Cheung, W. K. Chan, and Y. Yu, "Do maintainers utilize deployed design patterns effectively?" in *Proceedings of ICSE 2007*. IEEE Computer Society, 2007, pp. 168–177.

[63] P. Jablonski and D. Hou, "Aiding software maintenance with copy-and-paste clone-awareness," in *Proceedings of ICPC 2010*. IEEE Computer Society, 2010, pp. 170–179.

[64] T. H. Ng, S. C. Cheung, W. K. Chan, and Y. Yu, "Work experience versus refactoring to design patterns: a controlled experiment," in *Proceedings of FSE 2006*. ACM, 2006, pp. 12–22.

[65] D. S. Kirk, M. Roper, and M. Wood, "Identifying and addressing problems in object-oriented framework reuse," *Empir. Softw. Eng.*, vol. 12, no. 3, pp. 243–274, 2007.

[66] C. Oezbek and L. Prechelt, "Jtourbus: Simplifying program understanding by documentation that provides tours through the source code," in *Proceedings of ICSM 2007*. IEEE Computer Society, 2007, pp. 64–73.

[67] J. Stylos and B. A. Myers, "The implications of method placement on API learnability," in *Proceedings of FSE 2008*. ACM, 2008, pp. 105–112.

[68] L. Prechelt, B. Unger, M. Philippsen, and W. F. Tichy, "Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance," *IEEE Trans. Software Eng.*, vol. 28, no. 6, pp. 595–606, 2002.

[69] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers, "Improving API documentation using API usage information," in *Proceedings of VL/HCC 2009*. IEEE Computer Society, 2009, pp. 119–126.

[70] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Trans. Software Eng.*, vol. 31, no. 3, pp. 226–237, 2005.

[71] L. Layman, L. A. Williams, and R. S. Amant, "Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools," in *Proceedings of ESEM 2007*. ACM / IEEE Computer Society, 2007, pp. 176–185.

[72] M. Genero, M. E. Manso, C. A. Visaggio, G. Canfora, and M. Piattini, "Building measure-based prediction models for UML class diagram maintainability," *Empir. Softw. Eng.*, vol. 12, no. 5, pp. 517–549, 2007.

[73] E. Arisholm, H. Gallis, T. Dybå, and D. I. K. Sjøberg, "Evaluating pair programming with respect to system complexity and programmer expertise," *IEEE Trans. Software Eng.*, vol. 33, no. 2, pp. 65–86, 2007.

[74] F. Ricca, M. D. Penta, M. Torchiano, P. Tonella, M. Ceccato, and C. A. Visaggio, "Are fit tables really talking?: a series of experiments to understand whether fit tables are useful during evolution tasks," in *Proceedings of ICSE 2008*. ACM, 2008, pp. 361–370.

[75] M. M. Müller and A. Höfer, "The effect of experience on the test-driven development process," *Empir. Softw. Eng.*, vol. 12, no. 6, pp. 593–615, 2007.

[76] A. Bragdon, S. P. Reiss, R. C. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. L. Jr., "Code bubbles: rethinking the user interface paradigm of integrated development environments," in *Proceedings of ICSE 2010*. ACM, 2010, pp. 455–464.

[77] P. K. Linos, Z. Chen, S. Berrier, and B. O'Rourke, "A tool for understanding multi-language program dependencies," in *Proceedings of IWPC 2003*. IEEE Computer Society, 2003, pp. 64–73.

[78] L. C. Briand, M. D. Penta, and Y. Labiche, "Assessing and improving state-based class testing: A series of experiments," *IEEE Trans. Software Eng.*, vol. 30, no. 11, pp. 770–793, 2004.

[79] M. P. Robillard and G. C. Murphy, "Concern graphs: finding and describing concerns using structural program dependencies," in *Proceedings of ICSE 2002*. ACM, 2002, pp. 406–416.

[80] A. J. Ko, H. H. Aung, and B. A. Myers, "Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks," in *Proceedings of ICSE 2005*. ACM, 2005, pp. 126–135.

[81] L. C. Briand, Y. Labiche, H. Yan, and M. D. Penta, "A controlled experiment on the impact of the object constraint language in uml-based development," in *Proceedings of ICSM 2004*. IEEE Computer Society, 2004, pp. 380–389.

[82] D. P. Darcy, C. F. Kemerer, S. Slaughter, and J. E. Tomayko, "The structural complexity of software: An experimental test," *IEEE Trans. Software Eng.*, vol. 31, no. 11, pp. 982–995, 2005.

[83] Z. P. Fry and W. Weimer, "A human study of fault localization accuracy," in *Proceedings of WCRE 2010*. IEEE Computer Society, 2010, pp. 1–10.

[84] E. Arisholm, D. I. K. Sjøberg, and M. Jørgensen, "Assessing the changeability of two object-oriented design alternatives - a controlled experiment," *Empir. Softw. Eng.*, vol. 6, no. 3, pp. 231–277, 2001.

[85] M. Ceccato, M. D. Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "The effectiveness of source code obfuscation: An experimental assessment," in *Proceedings of ICPC 2009*. IEEE Computer Society, 2009, pp. 178–187.

[86] E. Aghayi, A. Massey, and T. D. LaToza, "Find unique usages: Helping developers understand common usages," in *Proceedings of VL/HCC 2020*. IEEE, 2020, pp. 1–8.

[87] M. Abi-Antoun, N. Ammar, and T. D. LaToza, "Questions about object structure during coding activities," in *Proceedings of CHASE 2010*. ACM, 2010, pp. 64–71.

[88] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *J. Softw. Evol. Process.*, vol. 25, no. 1, pp. 53–95, 2013.

[89] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, p. 3, 2007.

[90] F. Cuenca, J. V. den Bergh, K. Luyten, and K. Coninx, "A user study for comparing the programming efficiency of modifying executable multimodal interaction descriptions: a domain-specific language versus equivalent event-callback code," in *Proceedings of PLATEAU 2015*. ACM, 2015, pp. 31–38.

[91] T. Ishio, S. Kusumoto, and K. Inoue, "Debugging support for aspect-oriented program based on program slicing and call graph," in *Proceedings of ICSM 2004*. IEEE Computer Society, 2004, pp. 178–187.

[92] V. R. Basili and R. W. Selby, "Comparing the effectiveness of software testing strategies," *IEEE Trans. Software Eng.*, vol. 13, no. 12, pp. 1278–1296, 1987.

[93] R. R. Panko, "What we know about spreadsheet errors," vol. 10, pp. 15–21, 1998.

[94] M. Boshernitsan, S. L. Graham, and M. A. Hearst, "Aligning development tools with the way programmers think about code changes," in *Proceedings of CHI 2007*. ACM, 2007, pp. 567–576.

[95] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings of ICSE 2005*. ACM, 2005, pp. 117–125.

[96] A. J. Ko and B. A. Myers, "Source-level debugging with the whyline," in *Proceedings of CHASE 2008*. ACM, 2008, pp. 69–72.

[97] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: integrating web search into the development environment," in *Proceedings of CHI 2010*. ACM, 2010, pp. 513–522.

[98] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, "Back to the future: The story of squeak, a practical smalltalk written in itself," in *Proceedings of OOPSLA 1997*, vol. 32, no. 10. ACM, 1997, pp. 318–326.

[99] P. Rein, T. Beckmann, E. Krebs, T. Mattis, and R. Hirschfeld. (2023, May) Dataset for "Too Simple? Notions of Task Complexity used in Maintenance-based Studies of Programming Tools". [Online]. Available: http://doi.org/10.5281/zenodo.7632523