Toward Understanding Task Complexity in Maintenance-Based Studies of Programming Tools

Patrick Rein
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

Tom Beckmann
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
tom.beckmann@hpi.uni-potsdam.de

Toni Mattis
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
toni.mattis@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

ABSTRACT

Researchers conducting studies on programming tools often make use of maintenance tasks. The complexity of these tasks can influence the behavior of participants significantly. 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 make more deliberate decisions about the complexity of their tasks, we discuss different factors of task complexity. We draw these factors from previous user studies on programming tools as well as from a task complexity model from ergonomics research that we apply to maintenance tasks. 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.

CCS CONCEPTS

• **Software and its engineering** → **Software notations and tools**; • **Human-centered computing** → *HCI design and evaluation methods.*

KEYWORDS

task complexity, user studies, experiments, methodology

ACM Reference Format:
Patrick Rein, Tom Beckmann, Toni Mattis, and Robert Hirschfeld. 2022. Toward Understanding Task Complexity in Maintenance-Based Studies of Programming Tools. In *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming (‹Programming› '22 Companion), March 21–25, 2022, Porto, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3532512.3535223

‹Programming› '22 Companion, March 21–25, 2022, Porto, Portugal

ACM ISBN 978-1-4503-9656-1/22/03…$15.00
https://doi.org/10.1145/3532512.3535223

1 INTRODUCTION

Researchers working on programming tools empirically study programmers using tools through experiments or user studies [18]. 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" [12, 14]. 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 is prominently discussed in papers that describe strategies to design studies on programming tools [14, 18]. 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 [10, 31, 36]. As a result, the tasks may determine the explanatory power of an experiment or the usefulness of observations in user studies and their interpretation [14, 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 [10, 31]. For example, programmers may employ different debugging strategies when working on a small method in comparison to working on a large system [36]. 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 the 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 enable researchers in analyzing and shaping the complexity of maintenance tasks so that those are appropriate for their research questions. Therefore, we describe the factors that contribute to task complexity based on a comprehensive and generic framework of task complexity used in ergonomics research [23]. The framework describes 27 factors of task complexity summarized from previous research [23]. We customize this framework for maintenance tasks used in programming tool studies. To keep our framework focused, we limited our scope to studies with trained programmers. We then use our instantiated framework to determine and arrange specific complexity factors of maintenance tasks from related work and our experience. Our framework can be used to analyze purposefully designed tasks as well as tasks retrieved from existing projects.

In the following, we define task complexity, contrast it to related concepts such as task difficulty, and illustrate the role of maintenance tasks in studies on programming tools (Section 2). We briefly introduce the task complexity framework used and our adaptation to maintenance tasks for programming tool studies (Section 3). Within our adapted framework, we present factors making up task complexity (Section 4).

2 MAINTENANCE TASKS IN STUDIES AND TASK COMPLEXITY

Maintenance tasks are used in a variety of studies on programming tools. There are various kinds of maintenance tasks, but we focus on perfective and corrective tasks in this paper and introduce them briefly. To lay the foundation for the subsequent analyses, we define task complexity and contrast it to task difficulty. While task complexity is seldom use and defined in programming tools studies, we show examples of studies that already discuss some task characteristics that contribute to task complexity. Finally, we briefly point out how researchers in other researchers areas use and describe task complexity, namely in industry ergonomics and information retrieval.

2.1 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 observational studies as well as in controlled experiments [20, 33]. There are typically two forms of maintenance tasks: adaptive and corrective [7]. They are used by researchers in various kinds of studies [12, 14, 18] to evaluate tools, to test cognitive models, and to investigate programmer behavior [3, 20, 30]. In both kinds of maintenance tasks, participants receive an existing program or system and some description of the desired change.

In *perfective* maintenance tasks participants should adapt existing features or add new features according to some description of the new behavior [7]. 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 [20].

In *corrective* maintenance tasks, participants are asked to repair defective behavior [7]. 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 [6, 33]. 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 used two real defects in a large project [19]. Participants were asked to repair the defects.

2.2 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. Before this detailed discussion, however, we first define the term *task complexity*.

In this paper, we use a definition of task complexity that defines it as "the aggregation of any intrinsic task characteristic that influences the performance of a task" [23]. As a result, we regard task complexity as a compound concept that subsumes other properties of tasks [23]. The criteria that the characteristics have to influence the performance of a task are further 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" [23]. Thus, task complexity describes properties of the task that may influence generic task performers.

Task difficulty is related to task complexity. However, in contrast to task complexity, task difficulty depends on individual task performers [23]. For this work, we use the definition that task difficulty is the effort task performers perceive when working on a task [23]. Thus, task difficulty results from the combination of task complexity and the personal resources that specific task performers have. Examples of such personal resources in the context of programming tools are how experienced task performers are in programming and how much task performers know about the application domain [16, 32]. Researchers often use task difficulty to describe tasks they use in studies [14, 18] and commonly assess or shape task difficulty through expert judgment or piloting.

In summary, task complexity and task difficulty are both relevant for study designs. While task complexity is relevant to the actual research questions, task difficulty is also a practical concern for study designs. Matching task complexity and participants is a common challenge, in which analyzing task complexity can help determining sources of complexity that may become difficult for participants.

2.3 Task Characteristics in Studies Based on Maintenance Tasks

The tasks used in studies are a major influence on participants. As they determine how useful and generalizable observations are, researchers generally already discuss their tasks in detail via a variety of characteristics. Some researchers even do this in great detail. An example for such a description is an evaluation study of a new user

interface metaphor for integrated development environments [4, 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 debugging tasks [27, p. 1391]. Among the described characteristics are the length of the infection chain[1] of the defects, the presence of tests, and whether the defects are wrong or missing code.

These studies already describe several task characteristics that contribute to task complexity according to the definition we use. Nevertheless, as there is no structured guidance on task complexity, potentially relevant characteristics are often missing from the description of tasks. An example for 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 feature met the specification.

Some studies state task complexity as a characteristic that is relevant for their study without further describing the properties that make a task complex. [19, p. 1575]. For example, in an evaluation of a specialized back-in-time debugger, the description of tasks mentioned that the researchers selected tasks "that varied in complexity and difficulty". Beyond this statement, the nature of the task complexity was not explained any further.

2.4 Task Complexity in Other Research Areas

Task complexity is a common concept used to characterize tasks in other research areas, such as *information retrieval* [34] or *industrial ergonomics* [23–25].

In information retrieval research, task complexity is used to characterize search tasks [34]. Researchers studying information retrieval use task complexity in studies to evaluate search systems or observe usage patterns to inform theories of information retrieval. As put in one survey on task complexity in these studies, "a significant research challenge is developing tasks and task descriptions, such that the task itself can be systematically manipulated as part of the research design and does not become a confounding variable."

In industrial ergonomics research, task complexity is used to assess processes, such as continuous monitoring of a nuclear power plant [24, 25]. Researchers use task complexity to analyze the structure of the processes and spot potential complexity factors.

[1] The infection chain generally refers to the number of steps between the instruction that creates an erroneous run-time state and the instruction that leads to wrong surface behavior of the program [27, 36]. The length of an infection chain is measured in various different ways.

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

Table 1: Parts of the generic model of task components and their respective complexity-contributing factors (CCFs) [23]

Task components	CCFs
Goal/output	Clarity
	Quantity
	Conflict
	Redundancy
Input	Clarity
	Quantity
	Inaccuracy
	Redundancy
	Conflict
	Unstructured Guidance
	Mismatch
Process	Clarity
	Quantity of paths
	Quantity of actions/steps
	Conflict
Time	Pressure

3 ADAPTED TASK COMPLEXITY FRAMEWORK

Our framework is based on a generic framework of task complexity that is supposed to provide a generic perspective of task complexity independent of particular domains [23]. This generic framework is the result of a review of 24 previous models of task complexity. In the following, we describe our steps to translate this generic perspective to an adapted perspective for tasks in program maintenance studies.

Through this framework, we want to help researchers in reasoning about or shaping the complexity of their maintenance study tasks. At the same time, the framework does not provide guidance on other important considerations such as external validity, task difficulty, learning effects between tasks, or the duration of tasks. Neither does the framework present a comprehensive theory on actual software maintenance, focusing instead solely on the complexity of maintenance tasks used in studies.

3.1 The Task Complexity Framework

The generic framework our work is based on is the result of a review of models defining task complexity.

The generic framework provides a set of five components affecting task complexity. For each component, the authors define a set of complexity-contributing factors (CCFs) that are concrete factors that increase or decrease the complexity of its component, as shown in table 1. The factors are also abstracted into ten dimensions that are supposed to describe the phenomenon of task complexity in its most generic form.

For our adaptation of the framework, we focused on the components and their CCFs, as we found these to provide a more concrete taxonomy of complexity than the more abstract dimensions. The first component describes *goal and output* factors; its CCFs affect the abstract goal participants need to reach through for example

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* factors are determined by 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.

3.2 Adaptations for Tasks in Program Maintenance Studies

To identify sources of complexity in program maintenance tasks, we considered models of phases during a maintenance task [15, 28][22, p. 191]. According to a rough summarization of these models, programmers would begin with an initial comprehension phase, where materials such as a bug description, an observed fault, or similar are analyzed. Next, programmers would move on to a bug location phase with the goal of identifying relevant code in the software system. To do so, 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. Finally, a repair or patch is created that should eventually yield a correct program. Drawing from the debugging phases, we extracted distinct variation points that researchers can affect in their study setup: the task description, the software system, the infection chain or feature location, the patch participants are expected to create, and the tool environment.

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. Components of the task complexity model affecting the description concern the *goal and output* component, such as the clarity of the goal and the quantity of defects. The concrete written or oral task description introduces complexity through factors of the *input* and *presentation* components, for example the description's size or amount of guidance given.

System. Both in the comprehension and in the bug location phase, the system in which the defect or missing feature is located plays a major role. Factors of the *input* component such as the system's overall size, domain, or code quality will likely affect participants' ability to formulate and test their hypotheses. A very small system may make it too simple for participants to identify a root cause of a bug, while a very large system presented with little guidance may render the task too complex.

Infection Chain and Feature Location. During the bug location phase, researchers have to take another deliberate decision next to the decision for a system, which is the nature of the infection chain or how the feature to be adapted is distributed in the system. For example, a defect that no longer occurs upon observation [13, p. 33] is likely to introduce significant complexity as part of the conflict factor in the *goal* component. The goal's clarity may for example also be affected by how common the defect or requested feature is.

Patch. Once participants understood the defect or located the place where the requested feature can be added, they enter the phase of creating a patch to address the concern. The *goal and output* component characterizes complexity introduced through the size and nature of the patch, such as how many distinct places in the code base need to be changed.

Tool Environment. Not explicitly stated as part of the model, 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 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 in formulating hypotheses, while a debugger or a REPL can help in testing hypotheses. Factors important to the tool environment are primarily found in the *process* and *presentation* components.

Note that not all components and factors from the generic task complexity summary as listed in table 1 are included in our discussion. For example, we have excluded the *concurrency* factory in the *time* component, as we have not found studies that introduced complexity through concurrency in the sense that participants must handle multiple tasks simultaneously. For the factors that are mentioned we either found a study that was describing the corresponding factor or considered it important from our own experience, in which case we explain its relevance in detail.

4 TASK COMPLEXITY CHARACTERISTICS OF PROGRAM MAINTENANCE TASKS

Based on related work and our experience, we discuss factors that influence the complexity of tasks.

4.1 Task Description

The task description is the starting point for participants and thus may significantly influence how they proceed through the task.

The *goal* of participants reading the task description is to understand the task in the form of features to be implemented or defects to be repaired. The complexity of understanding the task is influenced by the *quantity* of sub-goals. For example, many studies use just one feature or defect at a time [19, 27]. This allows participants to focus on a single problem. At the same time, some studies give participants multiple features or defects at the same time. In one study participants received descriptions for five features to implement at once [29, p. 891]. In another study, participants received descriptions for ten features, five for one application and five for a second [35, p. 4]. 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 for the different sub-tasks.

When understanding the task the main *input* for participants is the written or oral explanation of the task. One complexity-contributing factor of the input is the *quantity* of the task description material, or simply put, the length of the task description. Both,

overly short and overly long descriptions can make a task description complex [23]. For synthesized tasks this is fully under the control of the researchers [29, p. 891][20]. But for tasks collected from existing projects, this might be a challenge, for example when the description of a feature also refers to numerous business use cases that are irrelevant for the isolated situation of the study.

The other central CCF for the task description input is its *clarity*. If the task description is unclear, whether accidentally or on purpose, participants may have to spend more time determining what the desired behavior should be or what the defective behavior looks like in detail. One aspect of clarity is whether the behavior is explicitly stated or whether participants need to derive it from additional material such as a standard. Another aspect is the ambiguity of the description. If participants can have more than one major interpretation of the described behavior, the subsequent investigation becomes more complex.

The ambiguity of the task description also touches on the factors of *guidance* in the task material. For many tasks, researchers want participants to work on investigating the overall system and the defect or feature at hand. Thus, ambiguity is a necessary component to not provide too much guidance on how to solve the task. For example, if the task includes an accidental hint on the class containing the root cause of a failure, participants might search for defect less extensively, as they might already find it using this unintentional hint. In contrast, some guidance can also make it feasible to design a study with a large system or with a large set of sub-tasks. For example, one study provided pointers to two relevant classes out of the 301 classes in the system [29, p. 891].

Finally, *redundancy* in the task description might reduce the complexity of understanding the task. For example, when the desired behavior is described through prose as well as manual test scripts or automated tests, participants can read on the task from different perspectives. Many studies provide programmers with a set of test cases accompanying the task description [27, 29].

4.2 System

The system is the backdrop in front of which participants work on the task and thus it is a major *input* for working on the task. 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.

One source of the complexity of the overall system is the *quantity* of elements in the system. Many studies report the overall size of the system under study through metrics such as lines of code, number of methods, number of classes, and number of packages [4, 20, 27]. Beyond this general measure, the complexity of the task also depends on the size of the portion of the system that is actually relevant to the task, as reported in some studies [4, 20].

Task complexity and thereby participant behavior is also influenced by the *clarity* of the source code. Programmers work differently when working on a method describing a complex algorithm, a class with tangled concerns, or a whole module with a consistent architecture [31].

With regard to *clarity* we distinguish between the underlying control flow and the overall clarity of the code. For the underlying control flow, the complexity of the control flow influences how well the code is understood. Ways to measure this are cyclomatic complexity or cognitive complexity [5]. Beyond this basic complexity, the overall quality of the code also influences how well the system and task is understood. "Spaghetti code" [13, p. 33] and bad code [28, p. 21] have both been brought up by developers when asked about reasons for difficult defects. The clarity of the system also involves 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 have to infer less to find code relevant for some application behavior [26]. While 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.

Related to the clarity of the system is the factor of *mismatches* that may occur between what participants expect the code to do and what it actually does. Examples include surprising uses of language features or confusing naming [14, p. 108 f.]. This kind of complexity can require participants to put more effort into understanding code in situations that are not relevant for the research questions.

Inaccuracy of the source code can also make the task more complex. By its very nature, code itself is an accurate description of the behavior of the system. However, some parts of the system may have been left out for brevity. This in turn might confuse participants when some parts of the system are given, but others are not. For example, in one study, the researchers removed non-trivial API from the code to reduce the knowledge required to understand the code [4]. While this reduces the required knowledge, it also limits the participants' ability to determine the behavior at the call sites of the API and requires them to make assumptions.

Redundancy in the description of the system behavior can help reduce the complexity of understanding the system. Redundant descriptions of system behavior such as high-level design documents can provide a different perspective of the system. For example, in one study, participants received the user manual of the system so that they could read about the surface behavior [29, p. 891]. Other studies reduced the influence from such documents by either not providing additional documents [4, 20] or by explicitly removing all such documentation from the source code [1, p. 3].

While redundant, additional information can be of help, *conflict* between documents describing the system behavior can increase the complexity of the task. A major source for such contradictions are comments in source code, in particular when existing projects are used. One study dealt with this issue by removing all comments from the source code [4]. Another study, consciously chose to use incomplete documents to introduce complexity to a task. In this study part of the solution to the task was using an undocumented special case of an otherwise documented mechanism [29, p. 891].

Finally, the complexity of understanding also depends on the knowledge required. This includes knowledge about the application domain, as well as knowledge about the technical mechanisms used in the system. Programmers use different strategies for understanding a system depending on whether they have prior knowledge about the application domain [9]. Most studies report the application domain of their systems [4, 17, 21]. Researchers can control how much 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 [14, p. 108]. One study explicitly wrapped all of these APIs to prevent this as a source of complexity [4, p. 2509].

4.3 Infection Chain and Feature Location

While all previous considerations are the same for perfective and corrective maintenance tasks, for this aspect we have to distinguish between the two kinds of tasks. In perfective maintenance tasks, participants determine the code sections that implement the feature or the general behavior that should be modified. In corrective maintenance tasks, participants determine the code section that includes the defect that should be repaired. In the following, we will first discuss factors contributing to the complexity of perfective tasks and then factors of corrective tasks. For both, the complexity stems from feature or the defect as the desired *outcome* of the activity.

Thus, in perfective maintenance tasks, we look at the complexity of the overall feature to be adapted by participants. A major source of this complexity is the *quantity* of distinct locations that are relevant for the feature to be implemented (combining feature location and impact analysis) [11]. In particular, task complexity is likely higher when the code of a feature is scattered. Thus, some studies limit the search space by telling participants which modules they should investigate or outright limit their ability to inspect other modules [8].

With regard to the *clarity* of the goal of locating the feature, the major source of complexity is how obvious it is to participants what the right location is to adapt the feature. Existing code that only needs to be changed introduces less complexity than code that needs to be extended or even replaced by participants.

For corrective tasks, we characterize the complexity of the whole infection chain. Again, a major factor is the *quantity* of elements in the infection chain. A long infection chain length has often been reported as an indicator of complex defects [13, 28]. The more statements are between the observable failure and the original defect in the code, the more effort is to required to determine the defect. Similarly, an increasing number of defects that contribute to a failure also increases complexity. Typically, studies using corrective tasks only use one defect per failure [20, 27].

Another major factor is the *clarity* of the defect. A first aspect of this is the origin of the defect. Participants may have to investigate a program differently when investigating a defect resulting from a wrong specification than a defect resulting from a programming error [36]. Similarly, the kind of defect can influence the overall task complexity. Participants may recognize more common types such as missing nil handling more easily than less common types, such as an accidental modification of meta-objects. Correspondingly, studies typically mention the types of defects participants worked on [4, 27].

Another aspect of the *clarity* is the distinction between defects of commission and defects of omission [2][14, p. 111]. 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 explicitly reduced by commenting out important code [27]. In contrast, defects of omission result from missing code. This in turn, is more complex to determine for participants, as they have to understand the code well enough, to realize that a statement is missing.

4.4 Patch

The patch is the set of modifications the participants propose to solve the maintenance task. As the participants work towards this patch, we regard the patch as an *outcome* and describe its complexity correspondingly.

The *clarity* of what the patch needs to contain depends on the number of different places that need to be modified. If only one place needs to be modified, the scope of the patch might be obvious to participants. However, if multiple places need to be modified, coordinating these changes adds extra complexity to the task. For example, one study reported on the number of files participants needed to modify to implement the required features [29, p. 891].

Further, sheer *quantity* of edits contained in the patch influences complexity. 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 some statement [27]. Other studies keep track of the size of the minimal or typical patch [4][29, p. 891].

While generating the patch, participants may face complexity arising from *conflicting* requirements. 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 instruction on what counts as *repairing* the failure. Similarly, 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. Again, clear guidance on this question can reduce complexity in this regard for participants. Finally, 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"[21, p. 363].

Finally, complexity may also arise from *redundancy* between valid patches. If there are several possible patches, and it is difficult for participants to decide which ones are valid, the task becomes more complex in that regard. Again, one solution is to avoid complexity from patch generation completely, by making the patch a minimal edit [27]. Another option is to make it explicit to participants that many different solutions are possible and that they are valid as long as they solve the original requirements [20].

4.5 Tool Environment

Tools necessarily play a major role in the study design. As the tool itself is being evaluated, most complexity introduced by tools should be inherent to the study. However, the tool to be evaluated will most likely still be embedded in an environment that involves other standard tools that may skew study results if not considered. Complexity introduced by these standard tools can be captured by factors of the *process* of using these tools to work on the task.

The *clarity* of the process may be affected by the means provided to reproduce the defect. Some studies provide a test runner with failing test cases as part of the tool environment [27][29, p. 891], providing participants a clear entry point.

The number of tools at the participants' disposal, the *quantity of paths*, may reduce complexity. For example, giving participants a variety of tools fit for different purposes may support their investigation of the system. Studies involving larger tool environments tend to report which of the standard tools where enabled, and which were disabled during the study, also to ensure that comparisons between different tool environments were valid [4, p. 2510][22].

Conflict in the use of tools could occur if participants are given access to tools that may obscure the defect or lead them on a wrong path. A misuse of tools can require recovery steps, for example if participants edit generated or read-only artifacts without noticing.

The *quantity of actions or steps* required to obtain runtime information may increase as well. For example, in a tool environment where participants are required to wait for a lengthy compilation step, they may have less feedback available than if for example a REPL is provided, both, for understanding the system as well as for verifying their patch. The means to obtain feedback could also affect task complexity: access to a debugger may render some tasks easier compared to a tool environments where users are limited to printf-debugging.

4.6 Overall Considerations

Finally, some components may influence task complexity beyond the single variation points and throughout the overall study design. For example, imposing and communicating a *time* limit on the study may lead participants to act differently if they believe to have enough time to consider more options. For example, one study asked participants to complete as many tasks as they can within 15 minutes [6, 33].

Throughout the study, researchers may decide to provide explicit *guidance* as input whenever they feel participants may have gotten stuck, thus reducing the previously assessed complexity in the moment. For example, in one study the researchers provided help on using the tools whenever participants had difficulties with their basic usage, thereby reducing the complexity in that regard [21].

Similarly, if any of the tools crashes during use, these constitute *non-routine events* that participants need to deal with, increasing the overall complexity of the task.

5 CONCLUSION

We presented a first version of a task complexity framework tailored towards 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 get 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 for their research questions.

By design our framework does not provide means to judge task difficulty, as this ultimately depends on the combination of the 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.

Finally, we would like to note that this is a preliminary version of the framework. Currently, the framework is mostly based on evaluation studies and thus over-emphasizes their perspective on task design. In the future, a thoroughly defined survey of task design in studies should lead to a more complete picture.

ACKNOWLEDGMENTS

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 449591262. We also gratefully acknowledge the financial support of HPI's Research School[3] and the Hasso Plattner Design Thinking Research Program[4].

REFERENCES

[1] Emad Aghayi, Aaron Massey, and Thomas D. LaToza. 2020. Find Unique Usages: Helping Developers Understand Common Usages. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2020, Dunedin, New Zealand, August 10-14, 2020*, Michael Homer, Felienne Hermans, Steven L. Tanimoto, and Craig Anslow (Eds.). IEEE, 1–8. https://doi.org/10.1109/VL/HCC50065.2020.9127285

[2] Victor R. Basili and Richard W. Selby. 1987. Comparing the Effectiveness of Software Testing Strategies. *IEEE Trans. Software Eng.* 13, 12 (1987), 1278–1296. https://doi.org/10.1109/TSE.1987.232881

[3] Deborah A. Boehm-Davis, Jean E. Fix, and Brian H. Philips. 1996. Techniques for exploring program comprehension. In *Workshop on Empirical Studies on Programmers 1996*. https://books.google.de/books?id=G2HflXT2tkYC&lpg=PA2-IA1

[4] Andrew Bragdon, Robert C. Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola Jr. 2010. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems, CHI 2010, Atlanta, Georgia, USA, April 10-15, 2010*, Elizabeth D. Mynatt, Don Schoner, Geraldine Fitzpatrick, Scott E. Hudson, W. Keith Edwards, and Tom Rodden (Eds.). ACM, 2503–2512. https://doi.org/10.1145/1753326.1753706

[5] G. Ann Campbell. 2018. Cognitive complexity. In *Proceedings of the 2018 International Conference on Technical Debt*. ACM. https://doi.org/10.1145/3194164.3194186

[6] Curtis Cook, Margaret Burnett, and Derrick Boom. 1997. A Bug's Eye View of Immediate Visual Feedback in Direct-Manipulation Programming Systems. In *Proceedings of ESP 1997* (Alexandria, Virginia, USA) *(ESP '97)*. ACM, New York, NY, USA, 20–41. http://doi.acm.org/10.1145/266399.266403

[7] Thomas A. Corbi. 1989. Program Understanding: Challenge for the 1990s. *IBM Syst J* 28, 2 (1989), 294–306. https://doi.org/10.1147/sj.282.0294

[8] Fredy Cuenca, Jan Van den Bergh, Kris Luyten, and Karin Coninx. 2015. 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 the 6th Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU@SPLASH 2015, Pittsburgh, PA, USA, October 26, 2015*, Thomas D. LaToza, Craig Anslow, and Joshua Sunshine (Eds.). ACM, 31–38. https://doi.org/10.1145/2846680.2846686

[9] Françoise Détienne. 2001. *Software design cognitive aspects*. Springer. http://www.springer.com/computer/swe/book/978-1-85233-253-2

[10] Françoise Détienne and Elliot Soloway. 1990. An Empirically-Derived Control Structure for the Process of Program Understanding. *International Journal of Man-machine Studies* 33, 3 (1990), 323–342. https://doi.org/10.1016/S0020-7373(05)80122-1

[11] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *J. Softw. Evol. Process.* 25, 1 (2013), 53–95. https://doi.org/10.1002/smr.567

[12] Alastair Dunsmore and Marc Roper. 2000. *A Comparative Evaluation of Program Comprehension Measures*. Technical Report EFoCS 35-2000. Department of Computer Science, University of Strathclyde.

[13] Marc Eisenstadt. 1997. My hairiest bug war stories. *Commun. ACM* 40, 4 (1997), 30–37.

[3] https://hpi.de/en/research/research-school.html
[4] https://hpi.de/en/dtrp/

[14] Dror G. Feitelson. 2021. Considerations and Pitfalls in Controlled Experiments on Code Comprehension. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*. IEEE, 106–117. https://doi.org/10.1109/ICPC52881.2021.00019

[15] David J. Gilmore. 1991. Models of debugging. *Acta Psychologica* 78, 1 (1991), 151–172. https://doi.org/10.1016/0001-6918(91)90009-O

[16] L. Gugerty and G. Olson. 1986. Debugging by Skilled and Novice Programmers. In *Proceedings of CHI 1986* (Boston, Massachusetts, USA) *(CHI '86)*. ACM, New York, NY, USA, 171–174. http://doi.acm.org/10.1145/22627.22367

[17] Donghwi Kim, Sooyoung Park, Jihoon Ko, Steven Y. Ko, and Sung-Ju Lee. 2019. X-Droid: A Quick and Easy Android Prototyping Framework with a Single-App Illusion. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology, UIST 2019, New Orleans, LA, USA, October 20-23, 2019*, François Guimbretière, Michael Bernstein, and Katharina Reinecke (Eds.). ACM, 95–108. https://doi.org/10.1145/3332165.3347890

[18] Amy J. Ko, Thomas D. LaToza, and Margaret M. Burnett. 2015. A Practical Guide to Controlled Experiments of Software Engineering Tools with Human Participants. *Empirical Software Engineering* 20, 1 (sep 2015), 110–141. https://doi.org/10.1007/s10664-013-9279-3 .

[19] Amy J. Ko and Brad A. Myers. 2009. Finding causes of program output with the Java Whyline. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, Boston, MA, USA, April 4-9, 2009*, Dan R. Olsen Jr., Richard B. Arthur, Ken Hinckley, Meredith Ringel Morris, Scott E. Hudson, and Saul Greenberg (Eds.). ACM, 1569–1578. https://doi.org/10.1145/1518701.1518942

[20] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Trans. Softw. Eng.* 12 (2006), 971–987.

[21] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. 2007. Program Comprehension As Fact Finding. In *Proceedings of ESEC-FSE 2007* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. ACM, New York, NY, USA, 361–370. https://doi.org/10.1145/1287624.1287675

[22] Thomas D. LaToza and Brad A. Myers. 2010. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 185–194. https://doi.org/10.1145/1806799.1806829

[23] Peng Liu and Zhizhong Li. 2012. Task Complexity: A Review and Conceptualization Framework. *Int. J. Ind. Ergon.* 42, 6 (2012), 553–568.

[24] Peng Liu and Zhizhong Li. 2016. Comparison between conventional and digital nuclear power plant main control rooms: A task complexity perspective, part I: Overall results and analysis. 51 (2016), 2–9. https://doi.org/10.1016/j.ergon.2014.06.006

[25] Peng Liu and Zhizhong Li. 2016. Comparison between conventional and digital nuclear power plant main control rooms: A task complexity perspective, Part II: Detailed results and analysis. 51 (2016), 10–20. https://doi.org/10.1016/j.ergon.2014.06.011

[26] Bertrand Meyer. 1997. *Object-oriented software construction*. Vol. 2. Prentice hall Englewood Cliffs.

[27] Michael Perscheid, Michael Haupt, Robert Hirschfeld, and Hidehiko Masuhara. 2012. Test-driven Fault Navigation for Debugging Reproducible Failures. *Information and Media Technologies* 7, 4 (2012), 1377–1400. https://doi.org/10.11185/imt.7.1377

[28] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2017. Studying the Advancement in Debugging Practice of Professional Software Developers. *Springer Software Quality Journal* 25, 1 (2017), 83–110. https://doi.org/10.1007/s11219-015-9294-2

[29] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. 2004. How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Trans. Software Eng.* 30, 12 (2004), 889–903. https://doi.org/10.1109/TSE.2004.101

[30] Xin Rong, Shiyan Yan, Stephen Oney, Mira Dontcheva, and Eytan Adar. 2016. CodeMend: Assisting Interactive Programming with Bimodal Embedding. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology, UIST 2016, Tokyo, Japan, October 16-19, 2016*, Jun Rekimoto, Takeo Igarashi, Jacob O. Wobbrock, and Daniel Avrahami (Eds.). ACM, 247–258. https://doi.org/10.1145/2984511.2984544

[31] Anneliese von Mayrhauser and A. Marie Vans. 1994. Comprehension Processes During Large Scale Maintenance. In *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994*, Bruno Fadini, Leon J. Osterweil, and Axel van Lamsweerde (Eds.). IEEE Computer Society / ACM Press, 39–48. http://portal.acm.org/citation.cfm?id=257734.257741

[32] Anneliese von Mayrhauser and A. Marie Vans. 1995. Program Comprehension During Software Maintenance and Evolution. *Computer* 28, 8 (1995), 44–55. https://doi.org/10.1109/2.402076

[33] E. Wilcox, John jr., Margaret Burnett, J. Cadiz, and Curtis Cook. 1997. Does Continuous Visual Feedback Aid Debugging in Direct-Manipulation Programming Systems?. In *Proceedings of CHI 1997* (Atlanta, Georgia, USA). ACM, New York, NY, USA, 258–265. http://doi.acm.org/10.1145/258549.258721

[34] Barbara M. Wildemuth, Luanne Freund, and Elaine G. Toms. 2014. Untangling search task complexity and difficulty in the context of interactive information retrieval studies. *J. Documentation* 70, 6 (2014), 1118–1140. https://doi.org/10.1108/JD-03-2014-0056

[35] Leon A. Wilson, Yoann Senin, Yibin Wang, and Václav Rajlich. 2019. Empirical Study of Phased Model of Software Change. *CoRR* abs/1904.05842 (2019). arXiv:1904.05842 http://arxiv.org/abs/1904.05842

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