

Unanticipated Progress Indication

Continuous Responsiveness for Courageous Exploration

Marcel Taeumel

marcel.taeumel@hpi.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Jens Lincke

jens.lincke@hpi.uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Robert Hirschfeld

robert.hirschfeld@uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

ABSTRACT

Scripting environments support exploration from smaller programs to larger systems. From original Smalltalk workspaces to modern Python notebooks, such tool support is known to foster understanding. However, programmers struggle with unforeseen effects from script execution, disrupting their thoughts. Unexpectedly long response times, in particular, cause frustration due to progress info not being provided automatically for ad-hoc scripting tasks. In Smalltalk systems, experienced programmers can interrupt an unresponsive environment to look for such info manually. We propose an automatic approach for progress indication, using a watchdog that periodically scans the stack of script workers for known heuristics to then derive task identity, label, and progress metrics. Using Squeak/Smalltalk as an object-oriented, single-threaded, cooperative scripting environment, we argue that simple heuristics for list enumeration or other patterns can (1) keep users informed while (2) leaving scripts untouched and (3) mostly retaining execution performance. We believe that *Unanticipated Progress Indication* will encourage programmers to experiment with library interfaces and domain artifacts more often, which will reduce their cognitive load for an expedient programming experience.

CCS CONCEPTS

• **Software and its engineering** → **Integrated and visual development environments**; *Scripting languages*; *Software prototyping*;
• **Human-centered computing** → **Interactive systems and tools**; *User interface programming*.

KEYWORDS

scripting, objects, Smalltalk, user interface, program comprehension, responsiveness, flow, exploration

ACM Reference Format:

Marcel Taeumel, Jens Lincke, and Robert Hirschfeld. 2024. Unanticipated Progress Indication: Continuous Responsiveness for Courageous Exploration. In *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming (<Programming> '24 Companion)*, March 11–14, 2024, Lund, Sweden. ACM, New York, NY, USA, 7 pages.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

<Programming> '24 Companion, March 11–14, 2024, Lund, Sweden

© 2024 Copyright held by the owner/author(s).

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM

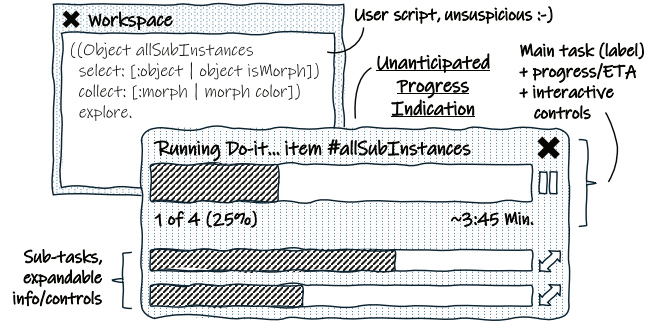


Figure 1: Even simple, unsuspecting scripts should inform users if expected response times would disrupt the flow of exploration. Descriptive progress indication can complement users' plans to move forward. While still blocking, an unresponsive system would frustrate users quickly, even if it would be just for half a minute. They do not want to be stuck in the unknown.

1 INTRODUCTION

What is happening? The system feels unresponsive. Is it stuck? Hm. Maybe endless recursion...or too much data? Will my script finish eventually? Maybe. Let's grab a coffee. Is it doing *anything* at all? I had just typed the following snippet of Smalltalk code into a workspace and evaluated it:

```
((Object allSubInstances
  select: [:object | object isMorph])
 collect: [:morph | morph color])
explore.
```

Just let me explore that list of colors! Hm. CPU load is high. That is good...right? Memory consumption is still moderate. Should I cancel the operation somehow? I could hit [CMD]+[.] and examine the debugger...or just wait a little bit longer? It is frustrating to not know what is going on in this little program. Exploration has been fun until now. Now my *flow* is broken. Better to plan ahead next time...huh. Maybe write some intermediate *progress info* to a log file? But what kind of info? How unfortunate. Why do I have to take care of this? I just want to explore and learn about the domain objects at hand, not experience technical inconveniences.

Eventually, the script finished after roughly 3:30 minutes. The programmer was frustrated by then. Later that day, she figured that there is a significantly faster way to enumerate all objects: SystemNavigation allObjects. Just about 200 milliseconds! And 1.5 seconds for the entire script. If only she knew before, flow might

have been retained. But trial-and-error is normal, mistakes will happen again. Progress indication would be comforting along the way.

Exploratory programming [2] is a practice where scripting thrives: low risk, high reward. Programmers are eager to explore problem and solution space with small code snippets, typically evaluated in a context full of domain-specific data. Experimentation is encouraged, because failure does not lead to anxiety but growth and understanding. There are scripting environments¹ for all kinds of general-purpose languages: Linux/Bash [11], Jupyter/Python [15], Lively/JavaScript [5], Squeak/Smalltalk [4]. Especially the latter, Smalltalk, has been known for its exploratory capabilities [13, 21] because of its garbage-collected object space, blending programming tools and user applications; a quite suitable (and forgiving) scripting environment with its powerful suspend-edit-resume debugger. We once tried to capture an experienced Smalltalker’s practice as follows [19]:

The exploratory mindset is an intensive form of user-to-software mediation where programmers are especially motivated to find a design that both works and inspires: “I know it when I see it.” Programming languages are expressive enough to unfold anything from unsurprising complexity to surprisingly beautiful simplicity. The latter is key.

It may be hard to learn and cherish because it is primarily about finding trade-offs: code readability, execution performance, possible nice-to-have features, minimizing dependencies while balancing maintenance cost [18], to name a few. Scripting is a valuable companion on this exploratory journey.

However, carefree experimentation can yield scripts that run longer than expected, leaving programmers uncertain about what is going on [20]. Frustration can set in quickly as “planning ahead” is not part of formulating and testing hypotheses during exploration. Instead, programmers expect a responsive system and (kind of) live feedback [12]. They want to remain informed to maintain task focus [9] and flow [1]. Especially when user interfaces support direct manipulation [3] of visual objects [6], which can be possible for arbitrary programming tools [17], programmers expect *some* response. Script execution typically starts with a key press or mouse click. Then what? For apparently simple scripts, we argue that waiting for 1–4 seconds is okay [14, p. 445], not breaking flow. Task complexity might raise this to 8–12 seconds, then some on-screen information would be nice. After 15 seconds, well, programmers might not even be sure whether the correct task is being performed or they made a mistake.

This observation yields the research question of this paper:

How can we keep programmers informed about the progress of *unexpectedly long-running* tasks without adversely interfering with their programming activities?

While background execution and/or (regular) progress indication is *the* answer for expectedly long-running tasks [10], our situation is peculiar for its single-threaded, memory-safe, simple characteristics. Programmers will not continue until a script has finished;

they are testing a specific thought. They are interested in a short response time and are even willing to optimize a complex routine, but only *after* they learned about its complexity. Thus, they are likely to deliberately ignore (or forget about) existing *push models* for progress indication in the system:

```
someTaskData
do: [:item | "costly operation" ]
displayingProgress: [:item | "info label" ].
```

Such extra code lines in between would obfuscate their script. They might not even consciously use a loop method such as `do:`, just a declarative expression like `allSubInstances` in the example above. Library providers, on the other hand, are most likely omitting such measures in low-level routines because there is no obvious connection to the user interface and it would slow down optimized algorithms (e.g., searching, sorting, enumerating).

We propose the concept of an *automatic watchdog* that is able to supervise a *script worker* to extract and show task progress using pre-defined *heuristics*. As a result, programmers will notice *Unanticipated Progress Indication*, even for low-level routines, as exemplified in Figure 1. Using Squeak/Smalltalk as an object-oriented, single-threaded, cooperative scripting environment, we argue that simple heuristics for list enumeration or other stack patterns can (1) keep users informed while (2) leaving scripts untouched and (3) mostly retaining execution performance.

In section 2, we explain the vocabulary around tasks and their progress, including how operating systems inform users through resource counters. Framing our contribution toward Smalltalk characteristics, we then describe the vices and virtues around suspend-edit-resume debugging for manual progress exploration in section 3. This paper’s main contribution lies in section 4, where we explain the mechanics of how the watchdog works, including selected heuristics for stack analysis. We discuss applicability and limitations in section 5. Finally, we conclude our thoughts in section 6.

2 TASKS AND (MISSING) PROGRESS INFO

A task processes a certain amount of items. It may be composed of sub-tasks working on other items. Such items belong to data structures such as lists, trees, and graphs. The work that is actually done depends on the situation and purpose of each task. For example, the sum might be aggregated from a list of numbers or the weights in a treemap might be accumulated from the leaves up to the root. Multiple complementing work is usually represented as composite tasks, with an outermost one as the starting point. That root task might be the entire script, just written by the programmer, waiting to be executed. From the outside, a script can look unsuspecting and small. Only a few lines of code. A quick response is expected. However, multiple hidden sub-tasks might be involved. Depending on the data items, such sub-tasks might take longer to finish than usual. Note that exploratory programmers do not only write scripts to verify assumptions or learn about an interface. They also add new code to the system and write their scripts against that code. Mistakes can happen along all such lines under control: endless recursion, memory leaks, long garbage-collection pauses, to name a few. If a script takes longer than expected, programmers want to be informed about its progress, or at least learn about the kind of sub-tasks involved.

¹Some programming languages feel indistinguishable from their execution environment and thus may not even flourish outside their usual “system” such as Smalltalk without an image or Bash without a shell.

Operating systems (OS) take care of their windows and processes. One such window might host the scripting engine,² taking longer than usual to refresh its view. If one window stops accepting incoming messages such as mouse clicks, it will be marked unresponsive and maybe visually faded into white. As illustrated in Figure 2, a so-called “Task Manager” can be used in concurrent or multi-processing environments to monitor tasks (or processes) from the outside. An unresponsive window can then be killed or restarted. More interestingly, there are *performance and resource counters* for each process such as CPU load, working-set memory, and open file handles. Depending on the script, experienced programmers might be able to derive a proper understanding of *task aliveness*. They want to decide whether it is worth waiting or time to kill. In rare circumstances, such counters might even indicate progress if they correspond to code in the script. For example, if an algorithm opens many files in the beginning and then closes them toward the end, the counter for open file handles might reflect that progress. In general, however, task managers are too coarse-grained to provide the expected level of detail so that programmers can actually understand what is going on in a long-running script.

Sometimes, programmers are able and willing to plan ahead. For example, if you copy a larger file in a command-line or GUI interface, you will most likely see a progress bar, maybe even a history of transmission speed, and an estimated time-to-finish. File-copy is a task with well-defined constraints and expectations. For a single small file, you will probably not even notice intermediate feedback. It will just be done immediately. While there are file-copy interfaces without progress indication, users learned that a copy operation can take its time, especially when floppy disks are involved. All in all, if programmers plan ahead, they will probably follow a *push model* that does not slow down the task noticeably:

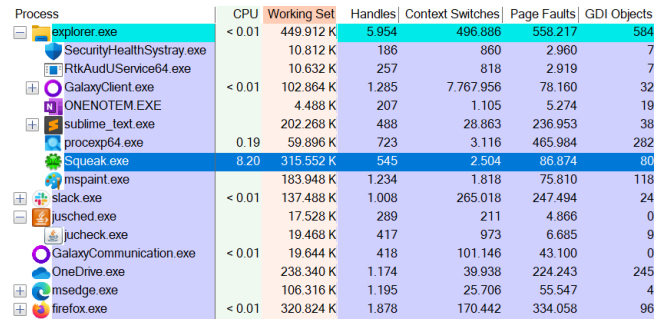
Push model Task workers report progress information periodically to a watchdog, which then visualizes³ that progress for the user. Workers do not have to reduce their update rate for performance reasons, only the visualization update should be efficient and not slow down the task. Yet, copying thousands of smaller files, for example, a command-line tool is usually much faster than the interactive GUI of the Windows Explorer, both showing progress.

Pull model Task workers do not report progress on their own. The watchdog will periodically extract information from the worker’s involved data structures. An example are resource counters (Figure 2), which can indicate task aliveness and maybe even approximate progress indirectly.

Still, scripting means ad-hoc experimentation, not planning ahead. And the designers of frameworks and libraries cannot proactively push progress info from every little enumeration involved in a low-level routine. Besides a dire performance impact, there would be many “false-positive tasks” (or micro-tasks) with meaningless descriptions, obfuscating the screen, rendering it even more

²There are engines (or runtimes) specific for profiling such as the VisualVM for Java (<https://visualvm.github.io>), which offer language-specific counters and maybe some insight into thread execution. For the scope of this paper, we assume that exploratory programmers are using their normal environment, unaware whether their next script would benefit from profiling or not. Our approach is lean and always active.

³In the simplest sense, one might just print dots to stdout to show task aliveness. So, the worker could directly write to a log file.



Process	CPU	Working Set	Handles	Context Switches	Page Faults	GDI Objects
explorer.exe	< 0.01	449,912 K	5,954	496,886	558,217	584
SecurityHealthSystray.exe		10,812 K	186	860	2,960	7
RtkAudUService64.exe		10,632 K	257	818	2,919	7
GalaxyClient.exe	< 0.01	102,864 K	1,285	7,767,956	78,160	32
ONENOTEM.EXE		4,488 K	207	1,105	5,274	19
sublime_text.exe		202,268 K	488	28,863	236,953	38
proceXP64.exe	0.19	59,896 K	723	3,116	465,984	282
Squeak.exe	8.20	315,552 K	545	2,504	86,874	80
mspaint.exe		183,948 K	1,234	1,818	75,810	118
slack.exe	< 0.01	137,488 K	1,008	265,018	247,494	24
jucheck.exe		17,528 K	289	211	4,866	0
jucheck.exe		19,468 K	417	973	6,685	9
GalaxyCommunication.exe	< 0.01	19,644 K	418	101,146	43,100	0
OneDrive.exe		238,340 K	1,174	39,938	224,243	245
msedge.exe		106,316 K	1,195	25,706	55,547	4
firefox.exe	< 0.01	320,824 K	1,878	170,442	334,058	96

Figure 2: Operating systems (here: Microsoft Windows) offer various performance and resource counters for each task (here: process). Users can monitor changes to draw rough conclusions about task aliveness. (Process Explorer 17.05, <https://sysinternals.com>)

irritating than a frozen window. Consequently, could there be a *pull-based approach* that is more fine-grained than what OS resource counters offer? Many scripting environments have simple rules for task scheduling: single-threaded, cooperative, maybe priority-based. Some of them provide advanced means for introspection, analyzing task state from within the environment, even down to the level of source code to fetch descriptive labels. In Squeak/Smalltalk, we found a promising contestant for such an environment that is able to foster carefree exploration and scripting, even for the not-so-experienced programmer.

3 SMALLTALK: ON-DEMAND DEBUGGER AS MAKESHIFT TASK INSPECTOR

Squeak runs in a single-threaded interpreter loop, typically within the OpenSmalltalk VM [8]. Concurrent behavior is realized through *green threading*, which manifests as *cooperative* scheduling of several Squeak processes [16]. *Preemption* is possible because processes have priorities, and processes can wait for (platform) synchronization objects such as semaphores. One important high-priority process is the *user interrupt watcher*. When the user hits [CMD]+[.], a certain semaphore is signaled, which wakes up that watcher, which then opens a debugger on the process that just got preempted [4]. For example, when a programmer types a code snippet into a workspace and then evaluates it, the single moderate-priority Morphic [6] process fires up the compiler and runs the resulting do-it method (or bytecode). Of course, a long-running script will block⁴ Morphic and thus make the system appear unresponsive.⁵ While it is actually running that script, no user input or other event handlers or animation timers will be processed until the script finishes. There is one⁶ exception: the user-interrupt shortcut [CMD]+[.],

⁴Yes, some scripts could run in the background. But the increased complexity for data synchronization interferes with low-effort, carefree exploration.

⁵There can be many reasons for a frozen Squeak. The system is self-sustaining. Programmers might thus break both applications and system tools. For example, putting a breakpoint into font-rendering routines will lock yourself out.

⁶Any unhandled error will also interrupt the script and result in a debugger. A new Morphic process will be spawned in this case to restore interactivity. The system could be “flooded” with debuggers this way, but there are selected precautions in place to avoid repeated execution of erroneous code.

which will preempt the Morphic process and present a debugger as depicted in Figure 3.

Inside the debugger, programmers can inspect all kinds of information around the suspended process. They have access to the entire stack, meaning all contexts or currently active methods. Each (reified) context object then refers to

- the receiver object of the call,
- the source code and program counter,
- objects behind argument names,
- objects behind temporary names if reached, and
- the next context.

Given that the script’s task and sub-tasks lie somewhere on this stack, programmers just have to figure out where to look. In Figure 3, a familiar `do:` is selected, which probably enumerates over some work items. The receiver is a collection, whose `size` is the goal, the current index the task progress. A context below, it reads `subclassesDo:`, which makes a somewhat descriptive label for a sub-task, explaining why Object `allSubInstances` takes so long. The entire class hierarchy is being enumerated to then fetch instances for each class separately; this is ineffective as it involves redundant filtering and unnecessary ordering. Anyway, the Smalltalk debugger acts as a *makeshift task inspector*. Programmers are able to navigate from script code to script process and back. Just like that. At any time.

Task (or process) information and context reification comes at a cost. The virtual machine does not maintain context objects during normal code execution, only for the debugging case. For example, just accessing the program counter *once* during a simple `3+4` can slow down the script 10x. And the garbage collector will also clean-up context objects later, which takes extra time. Luckily, accessing further stack frames does not increase the slow-down that much:

```
[3+4] bench.
'296,000,000 per second. 3.4 nanoseconds per run.
0.0 % GC time.'
[thisContext pc. 3+4] bench.
'26,300,000 per second. 38 nanoseconds per run.
10.4 % GC time.'
[thisContext sender sender sender pc. 3+4] bench.
'13,000,000 per second. 77 nanoseconds per run.
5.7 % GC time.'
```

This performance impact has implications for any watchdog that wants to supervise a process’ stack. The *sampling rate* should moderately reflect the actual need for details. Oversampling might negatively affect script execution in general. If programmers learn that “no info” means fast and “progress info” means slow, then we would have very little improvement over the regular on-demand debugger as a makeshift task inspector. They should not have to think about it but keep such a watchdog enabled all the time.

The reflective means that enable the Smalltalk debugger, namely stack inspection and code simulation, have “always” been enabling another powerful tool for profiling: the Message Tally. Programmers can spyOn: a code snippet, which will activate a watchdog that samples call stacks at a certain rate, producing a slice of the call tree as it could be observed. They can also tallySends:, which will employ code simulation to record the entire call tree as it happened. Spying is usually much faster compared to tallying, and it yields

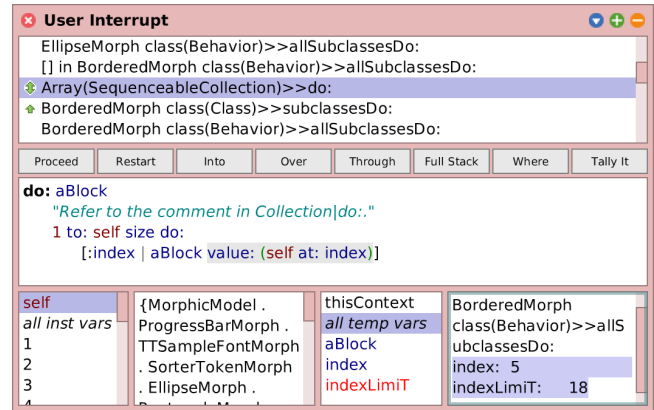


Figure 3: Users can interrupt a blocking process anytime via `[CMD]+[.]`. Stack frames reveal tasks and sub-tasks as well as their current progress. Yet, only experienced programmers know what to look for (here: `index` and `indexLimit`).

expedient information for most situations where programmers want to hunt down performance issues. It is a trade-off between level-of-detail and performance. Note that programmers benefit from keeping their exploratory flow, collecting information in an 80-20 manner. It does not have to be perfectly accurate. Good-enough is fine as long as it makes sense. *Can we provide such efficiency for progress indication?*

4 UNANTICIPATED PROGRESS INDICATION

We outlined the programmer’s exploratory mindset and technical constraints of a scripting environment as a baseline for our approach to Unanticipated Progress Indication. Now, we will explain the mechanics of an *automatic watchdog* that periodically analyses the stack of a *worker process* to yield an experience as sketched in Figure 1. To maintain the exploratory flow and curiosity, we require non-invasive means for (1) task identification and progress display as well as (2) task suspension, inspection, and resumption. Following a *pull model* or sampling approach, we thus cannot execute code before or after a task as we might miss such events. Any heuristic should also *minimize costs*, avoiding extra computation such as counting nodes in a tree when the structure has no such information readily available. Thus, some identified tasks will have a complete description while others won’t. We argue for the following selection of features:

Must-have Task identity (referring to worker stack), task label (referring to domain code), task display (referring to screen pixels) → *Unanticipated Task Indication*

Should-have Progress information (i.e., numbers for start, stop, and current position) → *Unanticipated Progress Indication*

Nice-to-have Estimated time-to-finish [7], maybe coordinated with sub-task estimations → *Toward Solving the Halting Problem [22] :-) for arbitrary scripts*

The interactive aspect of progress indication is basically for free: Smalltalk processes can be suspended, inspected, and resumed at any time. So, once informed, programmers can decide whether to

about a task to refine a script. Finding proper heuristics for task analysis, however, that is the main challenge.⁷

Our watchdog is configured with a set of *heuristics*, which it considers fully in every tick (e.g., every 250 milliseconds). Each heuristic exploits characteristics about the system’s frameworks or libraries, everything that might appear on the stack as in Figure 4. Such *generic* hints can be complemented with *specific* patterns, only found in particular parts of the application under construction. We focus on generic hints because programmers can rely on those for every new task they begin:

Task identity Find the context object that represents work to be done. It will be gone from the stack if the corresponding task is finished. Examples include loop methods or the beginning of recursions. In Squeak, a simple test for certain receiver classes (e.g., `Collection`) and message selectors (e.g., `#whileTrue:`) can be sufficient to avoid false-positives. Workspace do-its or `#bench` (see above) would require extra hints.

Task label From the identity context, find the next context object that points into domain code to then extract descriptive data that users can understand. Examples include package-name tests and simply showing the first specific message selector. In Squeak, the `printString` of a context can serve as task label. No need for costly code analysis. Identity contexts can be their own label such as for workspace do-its. They can also extend a generic label with *details about the current item* in the task if they have access to progress.

Task progress In (or around) the identity context, find numbers that represent steps in the task: first, last, current. Then the task display can derive the percentage done. Examples include lists with a growing index and the changing depth of recursions. Keep the percentage growing, and consider tail recursions to not do all the work at 100%. In Squeak, collections enumerate from 1 to their size, the current step is in a temporary called `index` or `i`. For lists of code statements, the growing program counter can indicate progress (`startpc`, `endpc`, `pc`).

Sub-tasks will occur when multiple task identities are found. In the single-threaded scripting environment, we expect that no heuristics find tasks in the lower part of the main (GUI) loop.

During the *lifetime* of each task, the watchdog will repeatedly query (and update) labels and progress info. A practical sampling rate is about every 250 milliseconds, directly connected with display update, resulting in 4 frames-per-second (FPS). Shorter tasks will not even be sampled, longer tasks can accomplish much work within that time. For simple tasks taking 1–4 seconds [14, p. 445], users will be informed about progress 3–15 times. Within the first 250 milliseconds, they might not even notice a hiccup given that they are in a working environment, not a game with elaborate animations targeting 60 FPS. After several measurements, the watchdog can start estimating the time-to-finish (ETA). Note that the time to process each item in a task might vary. Especially for arbitrary steps (or statements) in a script or heterogeneous collections, progress

⁷Note that, in this paper, we just provide hints on what interesting stack patterns might look like. We have no definite answer, not even for processes in Squeak/Smalltalk. In your own scripting environment, we suggest looking at how loops are represented. From there, other patterns might emerge.

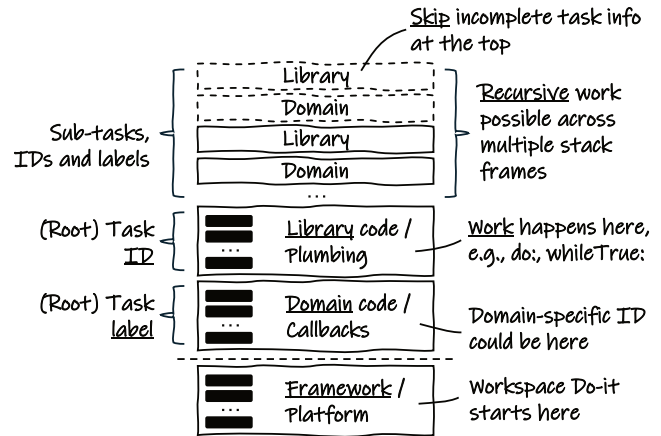


Figure 4: A process stack as data source for task analysis. Frames alternate between library code and domain code. Heuristics are defined for task ID and task label respectively. The number of frames analyzed can influence the overall performance. See Figure 3 for exemplary stack info. In Squeak, a stack frame is called “context object.”

does not have to be linear. We think that a sliding window of about 2–8 seconds can give somewhat reliable results for ETA, not being more detailed than “less than 10 seconds left”. Showing the elapsed time per task can further help users to decide whether it is worth to wait longer.

Having task labels and progress info (and maybe even ETAs), users finally expect to (1) see something and (2) interact with something as promised in Figure 1. Note that the watchdog is *not* the environment’s GUI process but a high-priority helper. In general, library interfaces and structures for the GUI might be reserved for a specific process to avoid data corruption. Luckily, also due to green threading, arbitrary Squeak processes can safely modify screen pixels and poll for user-input events [16], at least while Morphic itself appears unresponsive, evaluating a script. The watchdog *will not be preempted by Morphic* but should hurry to not slow down the script. So, the watchdog repeats the following steps in a loop:

- (1) Detect and update task information
- (2) Display task information on screen
- (3) Process user-input events (*optional*)
- (4) Wait for 250 milliseconds

To avoid visual glitches, the affected screen portion should be backed up and restored as sub-tasks appear and finish. Note that processing user input remains optional, because there is still `[CMD]+[.]` as a reliable keyboard shortcut to invoke the debugger. Yet, clickable buttons for task suspension, inspection, or cancellation offer a better affordance in a world of direct manipulation. Also note that the underlying host environment (or operating system) can provide additional resources to construct an interactive progress dialog. If possible, we think it is more immersive to remain within the scripting environment.

The watchdog can be notified explicitly about tasks and progress change. That is, our approach integrates (or can integrated with) existing push models such as Squeak’s `do:displayingProgress:`

mentioned before. If heuristics can identify such “push tasks” on the stack, scripts can push progress to the watchdog, displayed in the next analysis cycle. Both means complement each other because, sometimes, programmers can plan ahead (e.g., file-copy progress), and other times, they just want to dive into exploration and keep the flow (e.g., do-it progress). Indirect sampling heuristics might be more challenging to express efficiently, compared to a simple, direct notification directly in the source code. A trade-off remains. System maintainers can provide initial heuristics for the watchdog. Programmers might discover better ones during exploration. As tool building is part of exploratory programming in general, the definition of new heuristics can be considered tool building as well.

5 DISCUSSION

This section contains selected thoughts about applicability, usefulness, and limitations of our approach. While we would like to have such a feature in other scripting environments, we have only prototyped it for Squeak/Smalltalk, but with success. In other environments, scheduling constraints and overall process management might impose different rules for such progress watchdogs.

Implementation. As illustrated in Figure 5, we implemented a simple heuristic for collection enumeration, deriving a task label as described before and manually explored in Figure 3:

- For *task identity*, context objects of `Array>>do:` can be found and recalled on the stack during enumerations. Such contexts are *library code* in our model in Figure 4.
- For *task label*, each identity’s sender provides a descriptive print-string such as `Morph>>subclassesDo:`. Such contexts are *domain code* in our model in Figure 4.
- For *task progress*, each identity’s temporaries index and `indexLimit` represent *current* and *last*. *First* is always 1.

For the root task, the do-it method, we use the program counter as progress. The watchdog analyses up to 100 stack frames, but the depth while evaluating our example script was typically smaller. We could not measure a noticeable difference in execution time; it remained around 3:30 minutes \pm 5 seconds. However, there was a little bit more work for the garbage collector to do, probably due to the reified context objects from task analysis. Note that any other activity in the environment can impact script performance, because it is still single-threaded. Yet, we could test our assumption that a sampling rate of 250 milliseconds is a fair trade-off between task info and work load, at least with the heuristics we implemented.

Interference. There will always be some effects when a watchdog takes its time analyzing the stack of a worker process. The quality of the work will not be affected unless timing constraints are part of that quality. Examples include benchmarking routines such as `#bench` and `#timeToRun` in Squeak. While heuristics should be minimized, deeply recursive patterns on the stack might require a very high analysis depth to find all the tasks, meaning more context objects have to be reified and GC’ed later. If the window of analysis is too small, the actual root task can be missed, resulting in flickering on the screen as sub-tasks come and go. Interference can also happen between multiple high-priority watchdogs. For example, Message Tally (see above) might already be active, slowing

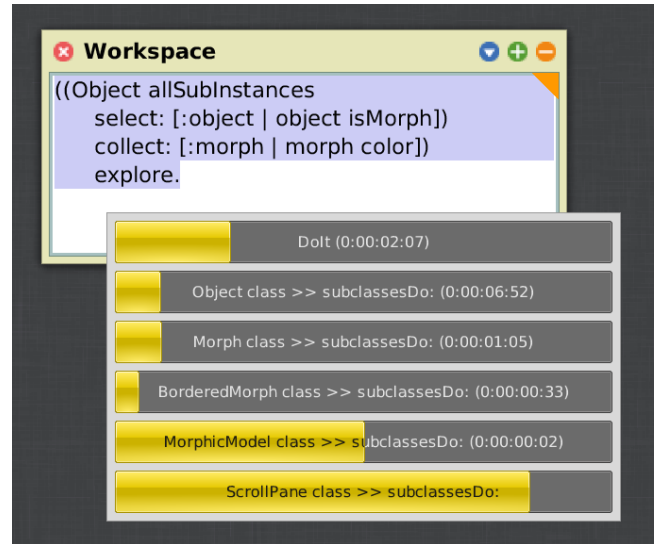


Figure 5: Unanticipated Progress Indication in Squeak, evaluating a Smalltalk script. Every 250 ms, a watchdog probes the UI process stack for collection enumerations, extracts progress metrics, and finds task labels in nearby contexts. We simplified the dialog as proof-of-concept but extra controls like in Figure 1 are possible. For interaction, a click on a progress bar will pause all tasks and open a debugger.

down the worker process even more. Some agreements between (high-priority) processes could help managing such choreography.

Quality. Some data structures maintain work items in a way that makes it challenging for a watchdog to extract task progress. Examples include streams where clients wait for a null byte or linked lists without index-based access. Actual numbers are required to calculate progress. Even iterative tree traversal will only maintain a sliding window of the next nodes, rendering it impossible with a sampling approach to understand the current position and the goal. Yes, the task itself can at least be identified and labelled. That is why we classified progress info as should-have, not must-have. An (updating) overview of active tasks and sub-tasks is added value in an otherwise unresponsive environment. For recursive algorithms, it depends on the data structure being processed whether progress can be derived. A simple list will probably shrink to nothing while a complex tree (or graph) is unlikely to show a meaningful, predictable pattern on the stack. At least the beginning of recursions can be used to identify (and label) tasks.

Maintenance. The programmer who maintains heuristics for libraries or frameworks is a tool builder, configuring the operations of the watchdog. Heuristics become part of the tool landscape, its code in need to be updated when dependencies change. We analyzed context objects and source code using representations and meta-information that might adapt in the future. For example, most enumerations result in a reference of `#whileTrue:` in the source code, but there is no actual message send in the corresponding byte code. It will be optimized by the compiler, a push here and a jump there. Luckily, the meta-data (here: literals) still contains

#whileTrue: as a symbol, which we query in heuristics. In general, similar-looking scripts can end up in different methods, which means that the corresponding names for temporaries can be different, which means that multiple heuristics are needed to extract task progress from loop methods:

```
(1 to: 10) asArray do: ... Collection >> #do:
(1 to: 10) do: ... Interval >> #do:
(1 to: 10 do: ...) (inlined)
```

Exploratory programmers should not be bothered to express scripts in a certain form that matches the existing heuristics. This would be comparable to following a push model, which they usually ignore in the first place. Instead, heuristics should be adapted (by tool builders) to cover more and more exploratory scenarios, increasing the robustness of Unanticipated Progress Indication.

Visual Glitches. An unresponsive system can be frustrating; a flickering or glitching progress dialog can be unsettling. As we discussed the quality of heuristics above, incorrect or unexpected task info can still face the user. Hinted in Figure 5, for example, the root task (or do-it) has an ETA smaller than its first sub-task. In our implementation, tasks do not negotiate their guesses; maybe they should. How would endless recursion look like? The screen would be filled with sub-tasks at some point. Why is a certain progress bar not growing but shrinking? The stop condition is probably changing, or the current position might be moving backwards; maybe a recursion-heuristic is broken. Why is that label so incomprehensible? Maybe because the corresponding domain method is just badly named. We think that watchdog heuristics cannot and should not compensate for bad code. They might hint at a refactoring that should be done. Who knows: *Can simple, effective task-progress heuristics even lead to better code quality in the software under construction?*

6 CONCLUSION

In a world full of unclear requirements and new applications for software every day, exploratory practices help programmers manage even steep learning curves in unfamiliar domains. Their tools and environments allow for experimenting and tinkering. Executable forms of explicated ideas, the program source code, are to be manipulated. “Scripting” is a skill, comparable with high-level programming. Programmers expect quick feedback for every script they evaluate. They make mistakes, but they will learn from iterations. Not everything works as envisioned; trade-offs must be found; results can nevertheless inspire and motivate. We presented Unanticipated Progress Indication as effective means that support this exploratory journey toward high-quality software, for an expedient programming experience.

ACKNOWLEDGMENTS

Sincere thanks go to all PX/24 reviewers, who provided valuable feedback by discussing this topic thoroughly. We are also thankful to Lukas Böhme, who kept asking interesting questions that helped clarify scope and details. We gratefully acknowledge the financial support of the HPI Research School “Service-oriented Systems Engineering” (<https://hpi.de/en/research-schools/hpi-sse.html>).

REFERENCES

- [1] Mihaly Csikszentmihalyi. 2008. *Flow: The Psychology of Optimal Experience*. Harper Perennial Modern Classics.
- [2] Richard P. Gabriel. 2014. I Throw Itching Powder at Tulips. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Portland, OR, USA). ACM, 301–319. <https://doi.org/10.1145/2661136.2661155>
- [3] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. 1985. Direct Manipulation Interfaces. *Human-Computer Interaction* 1, 4 (12 1985), 311–338. https://doi.org/10.1207/s15327051hci0104_2
- [4] Daniel H. H. Ingalls. 2020. The Evolution of Smalltalk: From Smalltalk-72 Through Squeak. In *Proceedings of the 4th ACM SIGPLAN History of Programming Languages Conference (HOPL IV)*. ACM, 1–101. <https://doi.org/10.1145/3386335>
- [5] Jens Lincke, Patrick Rein, Stefan Ramson, Robert Hirschfeld, Marcel Taeumel, and Tim Felgentreff. 2017. Designing a live development experience for web-components. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience* (Vancouver, BC, Canada). ACM, 28–35. <https://doi.org/10.1145/3167109>
- [6] John H. Maloney. 2002. *An Introduction to Morpich: The Squeak User Interface Framework*. Prentice Hall, Chapter 2, 39–67.
- [7] Joachim Meyer, David Shinar, Yuval Bitan, and David Leiser. 1996. Duration estimates and users’ preferences in human-computer interaction. *Ergonomics* 39, 1 (1996), 46–60. <https://doi.org/10.1080/00140139608964433>
- [8] Eliot Miranda, Clément Béra, Elisa Gonzalez Boix, and Dan Ingalls. 2018. Two decades of smalltalk VM development: live VM development through simulation tools. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (Boston, MA, USA). ACM, 57–66. <https://doi.org/10.1145/3281287.3281295>
- [9] Yoshiro Miyata and Donald A. Norman. 1986. Psychological Issues in Support of Multiple Activities. In *User Centered System Design: New Perspectives on Human-Computer Interaction*, Donald A. Norman and Stephen W. Draper (Eds.). Lawrence Erlbaum Associates, Inc., 265–284.
- [10] Brad A. Myers. 1985. The importance of percent-done progress indicators for computer-human interfaces. *ACM SIGCHI Bulletin* 16, 4 (1985), 11–17. <https://doi.org/10.1145/1165385.317459>
- [11] Eric S. Raymond. 2004. *The Art of UNIX Programming*. Addison-Wesley Professional Computing Series.
- [12] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding. *The Art, Science, and Engineering of Programming* 3, 1 (2018), 1:1–1:33.
- [13] Beau Sheil. 1998. *Datamation®: Power Tools for Programmers*. Morgan Kaufmann, Inc., Chapter 33, 573–580. <https://doi.org/10.1016/B978-0-934613-12-5.50048-3>
- [14] Ben Shneiderman and Catherine Plaisant. 2010. *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (5 ed.). Addison-Wesley.
- [15] Jeremy Singer. 2020. Notes on Notebooks: Is Jupyter the Bringer of Jollity?. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2020)*. ACM, 180–186. <https://doi.org/10.1145/3426428.3426924>
- [16] Marcel Taeumel and Robert Hirschfeld. 2016. Evolving User Interfaces From Within Self-supporting Programming Environments: Exploring the Project Concept of Squeak/Smalltalk to Bootstrap UIs. In *Proceedings of the Programming Experience 2016 (PX/16) Workshop* (Rome, Italy). ACM, 43–59. <https://doi.org/10.1145/2984380.2984386>
- [17] Marcel Taeumel and Robert Hirschfeld. 2021. Exploring modal locking in window manipulation: Why programmers should stash, duplicate, split, and link composite views. In *Proceedings of the Programming Experience 2021 (PX/21) Workshop* (Online, United Kingdom). ACM, 14–20. <https://doi.org/10.1145/3464432.3464433>
- [18] Marcel Taeumel and Robert Hirschfeld. 2022. Relentless Repairability or Reckless Reuse: Whether or Not to Rebuild a Concern with Your Familiar Tools and Materials. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2022)*. ACM, 185–194. <https://doi.org/10.1145/3563835.3568733>
- [19] Marcel Taeumel, Jens Lincke, Patrick Rein, and Robert Hirschfeld. 2022. A Pattern Language of an Exploratory Programming Workspace. In *Design Thinking Research: Achieving Real Innovation*. Springer, 111–145. https://doi.org/10.1007/978-3-031-09297-8_7
- [20] Marcel Taeumel, Patrick Rein, Jens Lincke, and Robert Hirschfeld. 2023. How to Tame an Unpredictable Emergence? Design Strategies for a Live-Programming System. In *Design Thinking Research: Innovation-Insight-Then and Now*. Springer, 149–166. https://doi.org/10.1007/978-3-031-36103-6_8
- [21] Jason Trenouth. 1991. A Survey of Exploratory Software Development. *Comput. J.* 34, 2 (1 1991), 153–163. <https://doi.org/10.1093/comjnl/34.2.153>
- [22] Alan Mathison Turing et al. 1936. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math* 58, 345–363 (1936), 5.