In Christoph Meinel and Larry Leifer (eds.).
Design Thinking Research: Innovation - Insight - Then and Now (pages 149-166)
Springer 2023 (doi.org/10.1007/978-3-031-36103-6_8)

How to Tame an Unpredictable Emergence?
Design Strategies for a Live-Programming System

Marcel Taeumel and Patrick Rein and Jens Lincke and Robert Hirschfeld

Abstract Programming environments that provide a *feeling of liveness* help professionals and amateurs alike to approach unfamiliar domains with ease through short feedback loops. Exploration and experimentation are promoted because any change to the program under construction can be observed immediately. However, live-programming systems such as Squeak/Smalltalk struggle with the predictable *emergence* of adapted program behavior as object communication can be unconstrained and diverse. While programmers wish for immediate effects, it would be helpful to at least know whether *anything* will happen after some time. In this chapter, we take a closer look at the means available in Squeak to explore and adjust object state and object behavior so that programmers can ensure the system's *responsiveness* and hence observe gradual or even induce eventual emergence. We argue that these *design strategies* are sufficient to architect communication patterns that reward changes with immediate effects. We believe that our work can help programmers to better understand their leverage toward a predictable emergence in systems whose *liveness* stems from *objects and messaging* in a space where tools and applications live side by side.

Marcel Taeumel
Hasso Platter Institute, 14482 Potsdam, Germany, e-mail: marcel.taeumel@hpi.uni-potsdam.de

Patrick Rein
Hasso Platter Institute, 14482 Potsdam, Germany, e-mail: patrick.rein@hpi.uni-potsdam.de

Jens Lincke
Hasso Platter Institute, 14482 Potsdam, Germany, e-mail: jens.lincke@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Platter Institute, 14482 Potsdam, Germany, e-mail: robert.hirschfeld@hpi.uni-potsdam.de

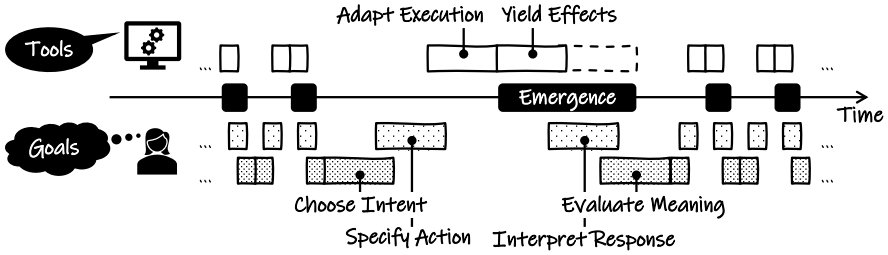

Fig. 1 The exploratory feedback loop: programmers translate their goal-oriented intents into actions that let the tool-driven system adapt the execution to then yield (observable) effects, which are to be interpreted and evaluated when planning for the next steps. A feeling of liveness occurs when the loop's frequency is high; direct manipulation [3] helps; continual responsiveness [11] is crucial. However, a timely emergence can be challenging to achieve [8].

1 Introduction

Programming feels empowering. We express our creative thought through digital media to experience and learn from new perspectives on our favorite domains. Artistic crafts, scientific simulations, engineering works: people can leverage the human intellect through computers. There are programming concepts, languages, and systems that can match – or challenge – our cognitive abilities. It can be intriguing (and fun) to explore a system's responses to our (creative) questions. But isn't programming hard to master? Maybe. Fortunately, the idea of *liveness* in a programming system makes programming more accessible – and exploration more efficient.

Exploratory programming is expedient for professionals and amateurs alike. Iteratively, any problem domain unfolds through experimentation; trial-and-error is encouraged for the sake of learning [10, 17]. That is, expert programmers are typically most curious toward unfamiliar challenges [1]; initially walking in the darkness like beginners, fumbling for light. Along the way, exploring the problem and solution space entails multiple conversations [14] with the programming system (and environment) at hand. Questions and responses. Input and output. Mouse clicks and animated graphics. Immediacy [18] plays an important role in this feedback loop [3]. If the system can answer in a timely fashion, programmers can safely follow their train of thought. Therefore, a feeling of liveness can boost the efficiency of exploratory practices [15, 14].

We consider a programming experience as live when the system takes only a few milliseconds to adapt its behavior and respond in a meaningful way. See Figure 1 for such an idealized feedback loop. In our setting, the act of live programming means continuously modifying a running system so that its observable effects immediately yield added value. Unlike other programming flavors, we focus on systems that allow for updating *executable software artifacts* in situ while avoiding restarts and loss of volatile information. Programmers still have to come up with an actionable plan, but the programming system at hand will not disrupt their train of thought with

unexpected pauses or unclear effects. Instead, all the tools and methods in the system will be like an unobtrusive extension of the mind.

The programming system of our choice is made of objects that communicate via messages [13]. This communication yields observable effects forming all kinds of object-oriented applications such as productivity tools, multimedia games, and educational simulations. Most notably, programmer tools and user applications share a single object space, which can expedite exploration (and experimentation) through short feedback loops. That is, tools allow for atomic updates of object identity, state, or behavior; the system will adapt immediately. Unfortunately, programmers might not notice (and understand) effects immediately as objects communicate concurrently and not always in an observable way. The objects in question might not even participate in any conversation at the moment and maybe they never will. Fortunately, our programming system – Squeak/Smalltalk[1] – provides several means to cope with such unpredictable emergence.

What does "emergence" mean in other domains? How can this concept help us understand the challenge of achieving liveness in programming (systems)? We ground our explanations on the following definitions:

> In philosophy, systems theory, science, and art, emergence occurs when an entity is observed to have properties its parts do not have on their own, properties or behaviors that emerge only when the parts interact in a wider whole. [...]
>
> — Wikipedia on "emergence" (2022-11-10)

> Emergence: the act or an instance of emerging.
> Emerging: newly formed or prominent.
>
> — Merriam-Webster on "emergence" (2022-11-10)

> The next phase comprises the emergence of an observable change in the behavior of the application from the adapted executable form. An observable change can be for example a changed textual output on the console, a different color of a graphical element, or a changed way of moving of a graphical element. [...]
>
> — Rein et al. on "emergence" [8]

As illustrated in Figure 1, we define "emergence" as the time between after a system's adapted execution and before a programmer comes up with the next idea on how to continue exploration. That is, we see liveness as primarily user-centered because observation, interpretation, and evaluation (of effects) all lead toward a better understanding. The system keeps on doing what it is currently programmed to do; the user (or programmer) sits nearby and perceives (visual or audible) output. A timely emergence can then lead to a feeling of liveness,[2] which makes exploration of problem and solution space more efficient.

Now, emergence becomes unpredictable not because a certain effect is supposed to show but because something different happens. Instead, any effect is supposed to happen and be seen immediately, or at least after some time [16]. That is the challenge:

---

[1] The Squeak/Smalltalk programming system, `https://squeak.org`

[2] Note that a feeling of liveness needs more than a timely emergence. A direct-manipulation interface [3] helps users to directly map their intents to actions (e.g., tangible object commands) as well as quickly figure out the meaning of what they can see (e.g., visual object representations).

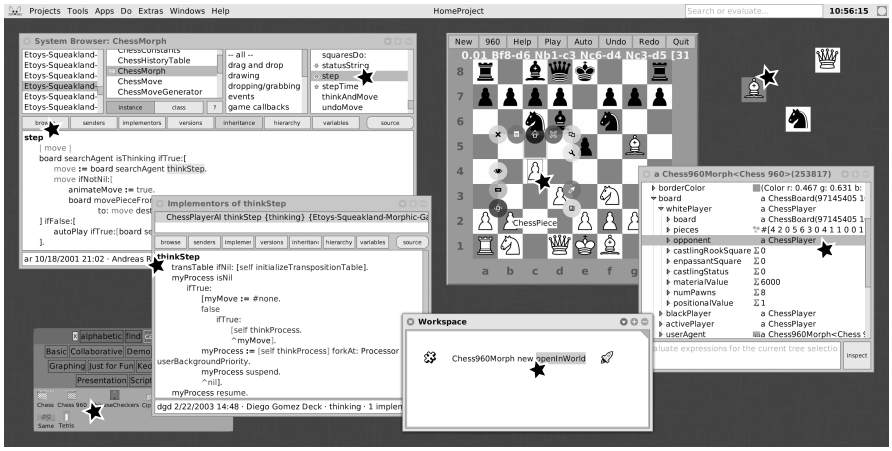

Fig. 2 The Squeak/Smalltalk programming system: tools and applications are made of objects, which communicate via messages to yield the desired behavior. Here, stars (★) denote such objects by example. Especially code objects have no visual representation other than (structured) text.

the Squeak/Smalltalk system (Figure 2) cannot guarantee a timely emergence because the objects of interest might not currently contribute to (observable) system behavior. The emergence is unpredictable in the general sense. Yet, there are predictable, domain-specific live-coding systems such as for audio or visual performance art [9], which leads us to our research question:

> How can we tame an unpredictable emergence in a general-purpose, object-oriented programming system to provide a feeling of liveness?

For simplicity, we assume that all other aspects of the feedback loop (Figure 1) are already optimized: (1) object-based design plan [6], (2) direct mapping of intent to action [3, 5], and (3) very fast adaptation [8]. What remains is the challenge after the system has adapted its execution: the unpredictable emergence.

Continual responsiveness[3] is crucial for uncovering gradual emergence in systems that struggle with predictability (and immediacy) for generic, unoptimized cases. That is, programmers can only use exploration tools within the system if one group of objects does not prohibit the communication in other important groups of objects. For example, mouse clicks should work or event timers fire even if a costly algorithm consumes precious computing time. Remember that objects represent tools and applications, and programmers use tools to inspect or modify these objects. Such

[3] Desirable response times depend on the particular use cases [11, p. 445]. Typing and cursor movement lies somewhere between 50 and 150 milliseconds. Simple frequent tasks such as button clicks should be no longer than 1 second until something happens. If it takes too long, users cannot make the connection between their action and the system's response. They might lose track of cause and effect.

means for exploratory programming allow for mitigating unpredictable emergence by manually "poking around" the objects that should be affected, maybe giving them an explicit "push" to see what happens. However, the tools will not work if the system becomes unresponsive. This goal of continual object communication leads us to our second research question:

> How can we design object communication to keep the system responsive to uncover gradual emergence or induce eventual emergence?

Note that our programming system offers general-purpose tools to create domain-specific applications. If it can remain responsive during divergent experiments, insights can converge into specific forms that offer immediate emergence during onward modification (or use). We are interested in laying out this design space, which enables a transition from gradual (or eventual) to immediate feedback. Programmers can then deliberately benefit from actual[4] *liveness* at some point when working on their project.

There is one more building block in Squeak to better fathom the potential of continual object communication: processes. Each process represents a communication thread where a chain of messages is to be resolved; objects in that chain are each waiting for a response; the one object at the end might just provide a response; or it needs to talk to another object first to do so. In programming terms, each process has a stack of activated methods, which grows and shrinks all the time. There are many processes in the system, but only a single one can run at a time.[5] To ensure fairness, scheduling depends on processes to give up computing time in a cooperative fashion. Then, the next process in line will get its turn; the current one goes back to the end. Additionally, each process has a priority and means to suspend its execution until a certain event (or semaphore) gets signaled. Processes with higher priority will always interrupt the ones with lower priority. Consequently, the system's responsiveness will be impaired if the process(es) responsible for user input and graphics output do not get their fair share of computing time.

In this chapter, we document design strategies that already exist in the Squeak/Smalltalk system or are provided by selected research extending that system. We try to generalize our findings toward continual object communication so that other objects-based systems might benefit from these ideas to improve their liveness as well. However, we are aware that our thoughts are biased – or too idealized – which could easily lead us

[4] To this day, the most promising systems for live programming offer level-4 liveness [16]. These systems are "informative, significant, responsive, and live." Program descriptions are human-readable and executable; modifying these descriptions changes system execution automatically without any noticeable delay. Systems with level 5 or 6 would be tactically and/or strategically predictive, but we do not know how such cleverness could be implemented. Squeak/Smalltalk provides level-4 liveness without immediate emergence [8], which, however, programmers can deliberately induce in their projects.

[5] Note that such green threading makes the virtual machine more robust because programmers cannot corrupt the object memory inadvertently. In an informal way, they can explore and experiment unless their experiments botch the processes (or means) responsible for continual responsiveness.

omitting challenges (or hard constraints) specific to other ecosystems. Nevertheless, we examine the following strategies:

Section 2 Programmers can write down and evaluate snippets of source code in almost any text field to explore the objects at hand.
Section 3 An extensible hierarchy of exceptions helps programmers understand and fix erroneous object communication.
Section 4 Code simulation enables a step-by-step observation of objects exchanging messages; time stands still to be explored and continued on demand.
Section 5 Users can hit a special key combination to interrupt an apparently unresponsive system to then explore it step by step.
Section 6 A graceful retreat from one interactive – yet unresponsive – framework to another – still responsive – one allows users to fix one "kind of liveness" with the help of another one.
Section 7 Watchdog processes can perform all kinds of process-monitoring tasks such as progress supervision, preemptive scheduling, and recursion detection.

Finally, in **Section 8**, we conclude our thoughts on how to tame the unpredictable emergence in the Squeak/Smalltalk live programming system.

2 Exploratory Workspace Everywhere

Programmers can type and evaluate code in any text field. The most prominent tool, which is just a text field plus window decorations, is the *workspace*. We documented several patterns around this conversational style with the system [14]. Yet, considering liveness and emergence, we focus on the basic mechanism that allows for talking to objects explicitly. Programmers might want to understand why a group of objects is or is not exhibiting a changed behavior. That is, they might want to observe gradual or induce eventual emergence.

Text (and source code) is a convenient and direct way of expressing one's thoughts. Users can type and easily revise their textual expressions; following an executable language helps the system understand the user. In Smalltalk, the syntax is straightforward almost like natural language:

```
anObject whatIsYourName.
anObject doSomething.
anObject do: #homework within: 60 minutes.
```

Here, we talk to `anObject` through three different messages. The first one, `whatIsYourName`, might only answer some text that helps us recognize a specific instance behind this rather generic "an object" label. Maybe it actually calls itself "a chess game." This would greatly improve our understanding and how to further interact with it. The second one, `doSomething`, might modify something in the system, maybe the object is "exhausted" afterwards, meaning its own state has changed as well. The third one, `do:within:`, appends arguments, which are objects

themselves. The first argument, `#homework`, is called (literal) symbol and hopefully means something to the receiving object. The second argument, `60 minutes`, is actually an example of another – albeit tiny – conversation: we ask the `60` to convert itself into a duration of sixty minutes. Both arguments hopefully mean something to the receiving object. All in all, programmers use this form of textual and direct communication for exploration in all kinds of programming tools.

Names are the tangible things programmers need to acquire before they can talk to objects. In technical terms, these names are *bindings* in a workspace. There is always the name "self" bound to an object in tools that allow for looking into that object itself. From the outside, programmers can come up and use any name that feels memorable and fitting:

```
| game pawn |
game := ChessMorph new.
pawn := game newPiece: 1 white: true.
```

Here, we declare, define, and use the names `game` and `pawn` for objects that represent a chess game. If we inspect the `game` to open a workspace (or inspector) that allows for looking into the game, we could then choose a more "intimate" way of talking to that object:

```
| pawn |
pawn := self newPiece: 1 white: true.
```

There can be all kinds of programming tools that bind the name `self` to any object of interest. Programmers must learn a tool's *scope*, which is usually indicated through visual cues. Multiple tool windows can each represent a scope (and namespace) of their own. Such a multi-tool usage supports programmers in following multiple ideas (or hunches) side by side, which reduces cognitive load and thus expedites exploration. Still, finding a good name can be quite a challenge.

If you do not know anything specific about an object, there are messages that all understand:

- Tell me your kind (or class).
- Describe yourself in brief text.
- Show me who is referring to you.
- Show me who you refer to.

There are objects that have a visual representation that might be better suited than (abstract) text. In recent Squeak, most graphical objects are *morphs* [5], which can be rendered per request to collect visual snapshots for pixel-wise examination. Note that the act of message passing can be observed (i.e., via "debug it") if a response is not satisfactory (i.e, via "print it" or "inspect it").

Through these simple means of direct communication, programmers can explore all objects in the system. Such an exploration might start with a visual handle or binding in a tool. Then, a simple "Tell me your kind" will reveal the object's class. A class can be used to browse the object's *vocabulary*. The correct vocabulary helps

form appropriate messages. Message arguments can still be challenging to find. However, object communication is often diverse and deep, and so is the exploratory journey toward improving the system's feeling of liveness.

3 Exceptions: Error, Notification, Halt

Exceptions bypass normal object communication to implement out-of-the-ordinary behavior. When an exception gets signaled, it takes a shortcut along the active chain of messages in a process. That is, the exception ignores the objects waiting for their responses and instead looks for an *exception handler* to talk to. That handler can then try to resolve the issue, abort the communication, or let the user intervene. Both exception and handler are objects themselves, which means that other objects must request their assistance in the first place. Whenever a problematic conversation might occur, one object requests a handler to watch out. Whenever a problem in a conversation does occur, another object requests an exception to go find the next handler. This mechanism, which is complementary to normal messaging, can help ensure responsiveness in the system.

The following example illustrates an ordinary payment process in a supermarket where a customer realizes that there is not enough cash left in the wallet:

```
OutOfCashException class >> signal
Customer >> lookIntoWallet       |
Customer >> doPayment            |
Cashier >> showReceipt           |
Cashier >> askForPayment         v
OutOfCashExceptionHandler >> watch
Cashier >> treatCustomer
...
```

Whatever the full chain of messages may be at this moment – maybe the customer's task is to buy milk and put it into the fridge at home – the top messages are the important ones. The cashier is prepared for this situation because it has happened before. Maybe the shopping cart can be parked until the customer returns with more cash later. The situation can be handled. The customer signals this exception as soon as the empty wallet is noticed. As a result, all communication down to `askForPayment` will be canceled. If the customer cannot otherwise proceed in this situation, the entire shopping process might need to be canceled. But what to put into the granola if not milk? In technical terms, objects can be in all kinds of unforeseen, troublesome situations, which then endanger the system's responsiveness.

There are three kinds of exceptions: errors, notifications, and halts. First, errors indicate that something unexpected has happened and that normal messaging cannot continue. An exception handler has to clean up if necessary; errors are usually not resumable. For example, `ZeroDivision` means that at some point the math has failed. While programmers can override and pretend that some other result is

meaningful, this might lead to even more errors along the way. Again, exploration and experimentation is encouraged, yet, risky when facing errors. Second, notifications (or warnings) indicate that something less serious has happened and can be ignored. An exception handler is not necessary but may be beneficial to better understand the cause of unexpected behavior. Notifications can be resumed. For example, `UnknownSelector` triggers an interactive dialog where the user can choose from known messages (or selectors) to let the source code compile. Third, halts are an exploratory tool that lets the communication stop at a well-defined point so that users can investigate the circumstances of unexpected system behavior. When `Halt` is signaled, the process will stop, an interactive debugger will open, and programmers can freely explore the state of affected objects. While time stands still for that process, programmers can communicate to the objects of interest and even cause changes.

A crucial instrument for reliable object communication (and a responsive system) is the error `MessageNotUnderstood`. In combination with an object's response to the message `doesNotUnderstand:`, programmers can interactively explore and repair erroneous processes. If the receiver is wrong, they can try to figure out which object should actually receive that message. If the message is wrong, they can change it to something more appropriate. However, if both receiver and message are basically correct, the programmer can choose to only implement the behavior and thus extend the object. The process will then resume, and the system hopefully exhibit the desired behavior. That is, `MessageNotUnderstood` and `doesNotUnderstand:` will promote short feedback loops. Programmers can thus experiment and send what they think is correct to the objects at hand.

There is a system-wide safety net in the form default exception handlers. That is, normal object communication does not have to use handlers at all, which can be tricky to foresee in most cases. The default effect of errors or warnings that are no handled is to signal an `UnhandledError` or `UnhandledWarning` exception. Now, the situation is codified as objects (or errors) again. If those special exceptions are not handled, they will simply stop the process and open a debugger. Then, programmers must intervene to fix the issue if possible and necessary. This safety net allows experiments such as evaluating `7/0` without being punished too seriously. Exploration is encouraged; short feedback loops can be established; an eventual emergence (of observable behavior) can be induced.

4 Stop and Step: Code Simulation

Message sends are very fast[6] thanks to the OpenSmalltalk VM[7] [7], which a promising baseline for message-based liveness and a responsive system. So, there is plenty of room for extra checks and safety nets that support carefree experimentation. As mentioned above, unhandled errors will suspend the process they were signaled in.

[6] On a Microsoft Surface Pro 6 with an Intel Core i7-8650U on Windows 10 21H2 in Squeak 6.0, the VM can process about 190,000,000 sends per second.

[7] The OpenSmalltalk VM, `https://opensmalltalk.org/`

A debugger window will appear, representing a tangible handle for that suspended process. Programmers can then explore the process' active methods[8] and continue message passing step by step. They might even be able to *go back in time*, which is not a feature of vanilla Squeak, but available as a research project.[9].

Code simulation is the mechanism that allows for understanding and fixing erroneous object communication as it happens. Live [4]. In contrast, code browsers merely show abstract pieces of text without a path to the running system. It is not obvious that any fast VM also offers a means to look into its execution details. For exploring communication patterns, however, we argue that it is necessary to "stop time" for selected processes. A suspended process can then be simulated, which means that message sending can continue with the help of other objects: the (method) contexts. Contexts provide access to the message receiver, active code, argument objects, and the object that is sending the current message (or active method). Like a puppeteer being in control from the outside, contexts can trigger a `step`, which lets the affected process "move forward." Programmers can then take their time to observe effects as they unfold.

In Squeak, there are three kinds of objects that reify system execution so that programmers can explore gradual or induce eventual emergence:

- A `Message` reifies an attempt of a message send and is passed as argument to `doesNotUnderstand:` in the case of a `MessageNotUnderstood` error.
- A `Process` reifies an active communication thread where objects are waiting "in line" to get responses from their neighbors.
- A `Context` reifies a piece of source code in execution, which is the implementation behind the message an object understands.

We will now use these means to reiterate the shopping experience as described above, but we will put extra cash into the wallet:

```
| process customer |
process := [ Customer new buyMilk ] newProcess.
[ process suspendedContext selector = #lookIntoWallet ]
    whileFalse: [ process step ].
customer := process suspendedContext receiver.
customer wallet add: 500.
process resume.
```

This simple example illustrates a very powerful idea: domain objects and meta objects can interact. From the outside in a workspace, we construct a process for buying milk. Then we simulate this process up to a point of interest, which is when the customer looks into the wallet. Then, we experiment by adding extra cash to the wallet. We

[8] Objects send messages. An object receiving a message will entail a message lookup, which means that a symbol will be matched to a piece of source code (or byte code). That code represents a method, which is an implementation of a message. That is why a suspended process has a stack of active methods, not messages. We use the term "chain of messages" to simplify this detail.

[9] Trace Debugger, a back-in-time debugger for Squeak, `https://github.com/hpi-swa-lab/squeak-tracedebugger`

can directly ask the process for the current context to then get access to the customer, which is the current receiver. Yes, we can also look into that customer's wallet; we could take things out of it, but here we add `500`. Finally, we let the (shopping) process resume. Is it enough cash? We will find out as the system execution continues. If necessary, we can always `suspend` the `process`, whose handle (or binding) we still have in our workspace.

Not all communication patterns can be understood through step-wise code simulation. For example, there can be two processes communicating through a shared resource, where one is adding and one is removing items. Suspending one process might block the other and vice versa. So, there should be tools that collect and present changes for groups of objects to reveal *steady frames* [2, p. 57]. What seems chaotic at the lowest level might oscillate in the middle or even be steady when observed from afar, like water drops escaping a garden hose as a constant stream. Considering object state, a data-driven perspective [12] might help programmers understand the flow of information. Considering object behavior, a higher-level debugger might help programmers understand larger patterns among processes. In sum, emergence does not only affect a few objects as it might go beyond sending a couple of messages in a single process. The VM is very fast and that thousands of messages can be exchanged within seconds. Liveness also implies a certain balance and harmony between all objects in the system.

5 Let the User Interrupt

Squeak's workspaces offer many ways to get the system stuck. Programmers experiment with small scripts to explore and poke around. Eventually, some attempt will have unexpected consequences and consume too much computing time, interfering with other processes that are waiting to be scheduled. Note that the system only "moves forward." Objects are created, communicate with each other, get modified, change behavior, and so on. There is usually no going back, which we call an *immutable past* [9].

What to do if there is no more user input possible? What if no code snippet can be typed and evaluated? How can the system be "restored" to a responsive state without losing information? Smalltalk systems have always had a simple answer for this situation: let the user interrupt the interfering process that is consuming all the computing time at the moment.

Even the most cautious programmers can get the system stuck by accident. For example, an experiment can become problematic by simply being mistaken about the (hidden) computing effort:

```
| n factorial |
n := 100000.
factorial := 1.
[ n > 0 ] whileTrue: [
    factorial := factorial * n.
```

```
    n := n - 1 ].
factorial explore. "... will start an object explorer ..."
```

Here, the problem is not necessarily in the algorithm written by the programmer: the factorial of 100000 took only about 5 seconds to compute. Instead, unpredictable effects can happen when other tools get involved: `factorial explore`. An *object explorer*, for example, will construct a textual representation of `factorial`, which might take some time for thousands of digits. Maybe there is more than one representation for that number object. Unoptimized tools might omit caching for costly inputs. Users cannot always anticipate this. The system seems to be stuck. But why exactly? This is generally not obvious. However, even a few seconds can be enough. Users can quickly get frustrated when confronted with an unresponsive system, let alone how it affects their desire for a feeling of liveness.

There is a dedicated keyboard shortcut to interrupt the active process. On macOS, for example, programmers can hit `[CMD]+[.]`, and a debugger will appear, representing the interrupted process. They can then explore and repair object communication like they would for unhandled exceptions. This reliable gesture should be invokable through a physical key, as the (virtual) system could be in any unclear state where pixels cannot be trusted anymore. Such an interrupt-key press then simply tells the active process to suspend:

```
| process |
process := Processor activeProcess.
process suspend.
```

Note that responsiveness will also be impaired if processes slow down the system. If basic interaction works – even if uncomfortable – programmers can use the *process explorer* to browse and suspend (or terminate) selected processes. Note that if tools work, programmers can also write scripts in workspaces to do the same thing. For example, they can query known process objects via `Process allInstances explore`. In general, given the class (object) of a domain concept, all domain objects can be fetched this way. There is only a single object space to explore and modify, where tools and applications exist side by side.

There are processes that must be running to keep the system responsive. They should not be interrupted as user input or graphics output would stop working, which are the basic means for responsiveness. A simple UI framework might look like this:

```
| uiProcess |
uiProcess := [ [
    InputEventSystem handleInputEvents.
    ScriptingSystem evaluateScripts.
    GraphicsSystem drawGraphics.
  ] repeat ] newProcess.
uiProcess resume.
```

An endless loop of input handling, script processing, and graphics drawing makes the system “look alive” and responsive. Provided that workspaces inject work into

`evaluateScripts`, programmers might inadvertently block (or slow down) this loop, which would affect input and drawing. Consequently, the system must ensure that another `uiProcess` will be started as soon as the current one gets interrupted. And somehow, that interfering script should be excluded from being evaluated a second time. Also, if an interrupted `uiProcess` is resumed again, the current one should be terminated to keep object communication in balance and harmony – too many cooks spoil the broth.

The correct timing of a user interrupt can be crucial for memory consumption. That is, too many objects might be created in a short amount of time. While the system looks unresponsive, object memory might "overflow." Fortunately, the OpenSmalltalk VM maps large process stacks to heap memory, which means that the usually limited stack memory is guarded so that users have more time to press that interrupt key. Still, such a memory leak can slow down the system even after the interfering process gets suspended. While obsolete objects get cleaned up all the time, incremental garbage collection can be essential to ensure responsiveness in a system with a large object memory. That is, the VM should always clean up "garbage" piece by piece without the user noticing.

6 Recursive Emergency

The objects that make up the *debugger* might stop working. That is, unexpected behavior (or erroneous communication) is not an issue limited to applications but extends to all kinds of things in the system, which includes programming tools. Recall that an unhandled error will suspend the process it was signaled in, which will then result in a debugger for this process. That debugger might as well signal another error when, for example, it tries to show itself on screen but somehow cannot. What happens when an unhandled error triggers another error that cannot be handled? We call this situation a *recursive error*. Recursion is the concept where a message is sent again while its response is being prepared. A recursive algorithm ends when a certain condition holds. For example, the Fibonacci sequence can be implemented as a method `fib:` that takes an integer argument `n` and then answers the result of `(self fib: n - 2) + (self fib: n - 1)`; the results for `n = 1` and `n = 2` are hard-coded to finish this computation. Now, how is a recursive error handled? Is there also an exit condition? But what to do when this condition holds?

A possible approach to handle recursive errors is to provide several distinctive UI frameworks as fallback [13]. Each framework has programming tools and thus means to modify the code base and all other kinds of objects. If one fails, the other one takes over. We assume that frameworks share as little source code as possible to increase the chances that an error does not carry over. In vanilla Squeak, there is Morphic by default and MVC as fallback. Both have debuggers, code browsers, and object explorers to the rescue. While both use common means, such as the code compiler, MVC has a much simpler architecture and is less likely to be extended at this time. Consequently, if an experiment in Morphic fails, tools in MVC can take over. There

can be programming tools even simpler than MVC, which we demonstrated through the keyboard-only SqueakShell [13].

The system detects a recursive error by flagging the erroneous process, which indicates that a debugger is about to open. If the debugger fails to open, that recursion flag will be detected on the second attempt and then trigger a transfer to another framework. Since every "thing" is an object in Squeak, frameworks are as well. They are represented as projects [13], which are a way to organize work and describe the basic means of responsiveness. For example, MVC uses a new process per tool for handling user interaction and graphics updates; Morphic (re-)uses a single UI process for all tools. When transferring the control from one project (or framework) to another, the erroneous process will be suspended immediately to not cause further harm. Programmers can then use the tools available in that other project to restore responsiveness as desired. As switching between frameworks can be tedious, we assume that there is only one preferred one where liveness matters.

If all else fails, there will be the *emergency evaluator*. With only a few lines of code, the system can provide a very primitive read-evaluate-print loop. Programmers can type expressions like they do in workspaces. For simplicity, there is no support for multiple lines, text selection, or bindings. What *must* work at this point, is keyboard input, text rendering, and the code compiler. Here is the basic idea (without output) of the emergency evaluator in pseudo code:

```
| line char |
[   line := ''.
    [ [ Sensor keyboardPressed ] whileFalse.
    (char := Sensor keyboard) = Character cr]
       whileFalse: [ line := line, char asString ].
    line = 'exit'
] whileFalse: [
    line = 'revert'
        ifTrue: [ System revertLastChange ]
        ifFalse: [ Compiler evaluate: line ] ].
```

The most important feature is to revert the latest code change. Chances are, that erroneous object behavior is the culprit. Since code (or behavioral) changes are versioned, reverting those is easy. However, state changes are not versioned. So if an object has unfortunate state, programmers must figure out a way to access that object and restore it manually. Recall that `allInstances` is a very powerful query mechanism to access any object provided that you know the name of its class.

7 Reliable Watchdogs

One processes can watch over another process. That is, programmers can design processes that *monitor* responsiveness in the system and intervene if necessary. Recall that Squeak's processes schedule cooperatively at the same priority level

and preemptively between priorities. Thus, for example, a process with priority 75 could wake up every 250 milliseconds to then monitor any process below or equal to 75. Processes under inspection will not change on their own but "stand still" until the watchdog yields control (to other watchdogs) or suspends for another 250 milliseconds. Recall that one process can easily access other processes in the system. So, our *watchdog* could log current receivers, depth of message stack, or other information that might be of interest. Note that the power of watchdogs is not constrained to monitoring. All objects are free to send any message to any other object they have access to. This freedom entails all kinds of possible side effects such as process termination, stack manipulation, and actually fiddling around with domain-specific objects in your application. A common example for a watchdog is the implementation of time-outs for test runs.

We will now describe four useful watchdogs, which each try to perform their analysis (and intervention) as quickly as possible to not slow down the system. Their wake-up time (or frequency) denotes the precision of their work. Consequently, watchdogs that need more time should wake up less often to not further jeopardize responsiveness in the system.

Call tracing and profiling. Squeak offers a sampling-based tool to trace object communication, which is called *message tally*. With a high frequency, a watchdog collects the stack of the tallied process and approximates a *call tree* from all samples. Programmers can then explore the most prominent communication patterns in the tally. The overall execution time can be used to estimate the duration of selected message sends. Note that sampling-based tracing trades performance for accuracy. Brief conversations between objects will likely not be traced as process scheduling does not allow for such fast sampling rates.

Unanticipated progress indication. There are cases where programmers are willing to wait in front of an unresponsive system so that maximum computing time is used to finish a workload as quickly as possible. However, they do wish for progress indication to plan their next steps. If known upfront, such indication can be requested:

```
workItems
    do: [:workItem | workItem process ]
    displayingProgress: [:workItem | workItem label].
```

Unfortunately, there can be numerous places in the systems where workload can spike unexpectedly, meaning that there is likely no progress indication in place. Also, a few long-running takes need a different treatment than thousands of short-running tasks. That is, any overhead in progress computation and display must be minimized. After all, programmers expect work to finish as fast as possible. Now, watchdogs can be used to analyze the process stack and look for such loops where work items are enumerated:

```
| workBlock start stop index |
workBlock := [:step | ... ].
index := start.
```

```
[ stop >= index ]
            whileTrue: [ workBlock value: index.
                         index := index + 1 ].
```

Given that a process exposes its current stack, a watchdog can access bindings such as `start` and `stop` and `step` to reveal progress. Sampling can codify "tasks" through methods that remain active after repeated observations. Programmers are not required to prepare such progress analysis for specific cases. The watchdog can use generic patterns to find tasks. As soon as it finds a task, it can tell the user about task progress. This indication is not anticipated but welcomed.

Infinite recursion detection. Watchdogs can find patterns that indicate *infinite recursion* in a process, which might indicate potential out-of-memory issues. The user should be informed that the process in question will be suspended for inspection. Yet, it might be challenging to distinguish a deeply nested algorithm that uses backtracking (or dynamic programming) from actual infinite recursion. As false-positives happen, users should always be in charge of deciding whether or not to abort a computation. Too many false-positives might disturb the programming experience up to a point where the watchdog will be disabled for good. Consequently, such background assistance should be configured to consider domain-specific workload and personal preferences. Still, generic patterns are a good starting point.

A time-slicing scheduler. Fairness in Squeak's process scheduling depends on the cooperation of processes running at the same priority level. However, a programmer's experiments might not be so cooperative after all:

```
[ anObject doExpensiveWork ] fork. "... in the background?"
```

Here, the programmer might think that performing an expensive task in another process keeps the system responsive. Yet, for example, in Morphic, all workspace scripts are evaluated in the same UI process. This means that the programmer either explicitly puts that task in a process with lower priority or slices it up with an occasional `Processor yield` in between. Now, a time-slicing scheduler could also help with this issue, introducing preemption within a priority level as well. Each process could just get some milliseconds before going back to the end of the line. Similar to progress indication, a watchdog can reconfigure process objects as required. Fairness might be such a requirement to ensure responsiveness in the system.

8 Conclusion

Admittedly, Squeak/Smalltalk is not the most convenient live-programming system. Its building blocks, objects and messaging, are expressive and powerful. The system is for generic purposes by design; there are little constraints to object communication. An uninformed programmer might get frustrated when some experiment gets the system stuck. We argued that programmers must know their tools and learn about

the means for exploration [15, 14] to get the system "unstuck" again. Responsiveness is the key to observing gradual and inducing eventual emergence.

Processes are the means for (de)composing elaborate communication patterns. Each process represents an active chain of messages, a topic, a discussion, a relevant conversation to be understood. You can type and evaluate snippets of source code in workspaces everywhere in the system. Chances are that the tool in front of you – maybe another debugger – provides bindings for objects of interest. Use an object's name and ask away; learn more about it. It might tell you why some effect is or is not showing up. Some messages will even help you fix an erroneous communication.

There are tool-building frameworks that provide specific constraints for guiding adaptation and emergence. For example, Morphic [5] offers the message `step` to codify recurrent actions for all graphical objects. Programmers (should) know that changing code in a `step` method will be executed at some point. For another example, Vivide [12] describes exploration tools as rules of data transformation and visual mappings. Programmers can then expect and rely on certain effects when modifying these rules. We think that frameworks on top of objects and messaging represent clear learning objectives. Once understood, programmers can express their goals in framework-specific actions to then rely on immediately observable effects. The Squeak/Smalltalk live programming system is thus capable of providing a predictable emergence for application-specific domains.

Acknowledgements Many thanks to Dr. Sharon Nemeth for her editorial support. We gratefully acknowledge the financial support of the HPI Research School on Service-oriented Systems Engineering (`www.hpi.de/en/research/research-schools`) and the Hasso Plattner Design Thinking Research Program (`www.hpi.de/en/dtrp`).

References

1. Gabriel, R.P.: 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, pp. 301–319. ACM (2014). DOI 10.1145/2661136.2661155
2. Hancock, C.M.: Real-time programming and the big ideas of computational literacy. Ph.D. thesis, Massachusetts Institute of Technology (2003)
3. Hutchins, E.L., Hollan, J.D., Norman, D.A.: Direct manipulation interfaces. Human-Computer Interaction **1**(4), 311–338 (1985). DOI 10.1207/s15327051hci0104_2
4. Ingalls, D.H.H.: The evolution of smalltalk: From smalltalk-72 through squeak. In: Proceedings of the 4th ACM SIGLAN History of Programming Languages Conference (HOPL IV), pp. 1–101. ACM (2020). DOI 10.1145/3386335
5. Maloney, J.H.: An Introduction to Morphic: The Squeak User Interface Framework, chap. 2, pp. 39–67. Prentice Hall (2002)
6. Meyer, B.: Object-oriented Software Construction, 2 edn. Prentice Hall (1998)
7. Miranda, E., Béra, C., Boix, E.G., Ingalls, D.H.H.: 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, pp. 57–66. ACM (2018). DOI 10.1145/3281287.3281295
8. Rein, P., Lehmann, S., Mattis, T., Hirschfeld, R.: How live are live programming systems?: Benchmarking the response times of live programming environments. In: Proceedings of the

Programming Experience 2016 (PX/16) Workshop, pp. 1–8. ACM (2016). DOI 10.1145/2984380.2984381
9. Rein, P., Ramson, S., Lincke, J., Hirschfeld, R., Pape, T.: Exploratory and live, programming and coding. The Art, Science, and Engineering of Programming **3**(1), 1:1–1:33 (2018). DOI 10.22152/programming-journal.org/2019/3/1
10. Sheil, B.: Datamation®: Power Tools for Programmers, chap. 33, pp. 573–580. Morgan Kaufmann, Inc. (1998). DOI 10.1016/B978-0-934613-12-5.50048-3
11. Shneiderman, B., Plaisant, C.: Designing the User Interface: Strategies for Effective Human-Computer Interaction, 5 edn. Addison-Wesley (2010)
12. Taeumel, M.: Data-driven tool construction in exploratory programming environments. Ph.D. thesis, University of Potsdam, Digital Engineering Faculty, Hasso Plattner Institute (2020). DOI 10.25932/publishup-44428. URL `https://doi.org/10.25932/publishup-44428`
13. Taeumel, M., Hirschfeld, R.: 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, pp. 43–59. ACM (2016). DOI 10.1145/2984380.2984386
14. Taeumel, M., Lincke, J., Rein, P., Hirschfeld, R.: A pattern language of an exploratory programming workspace. In: Design Thinking Research: Achieving Real Innovation, pp. 111–145. Springer (2022). DOI 10.1007/978-3-031-09297-8_7
15. Taeumel, M., Rein, P., Hirschfeld, R.: Toward patterns of exploratory programming practice. In: Design Thinking Research: Translation, Prototyping, and Measurement, pp. 127–150. Springer (2021). DOI 10.1007/978-3-030-76324-4_7
16. Tanimoto, S.L.: A perspective on the evolution of live programming. In: 2013 1st International Workshop on Live Programming (LIVE), pp. 31–34. IEEE (2013). DOI 10.1109/LIVE.2013.6617346
17. Trenouth, J.: A survey of exploratory software development. The Computer Journal **34**(2), 153–163 (1991). DOI 10.1093/comjnl/34.2.153
18. Ungar, D., Lieberman, H., Fry, C.: Debugging and the experience of immediacy. Communications of the ACM **40**(4), 38–43 (1997). DOI 10.1145/248448.248457