Evolving User Interfaces From Within Self-supporting Programming Environments

Exploring the Project Concept of Squeak/Smalltalk to Bootstrap UIs

Marcel Taeumel* Robert Hirschfeld* †
*Software Architecture Group, Hasso Plattner Institute, University of Potsdam, Germany
†Communications Design Group (CDG), SAP Labs, USA; Viewpoints Research Institute, USA
marcel.taeumel@hpi.de robert.hirschfeld@hpi.de

ABSTRACT

It is common practice to create new technologies with the existing ones and eventually replace them. We investigate the domain of user interfaces (UIs) in self-supporting programming environments. The Squeak/Smalltalk programming system has a history of almost 20 years of replacing Smalltalk-80's model-view-controller (MVC) with Self's Morphic, a direct manipulation interface. In the course of this transition, we think it is likely that Squeak managed to provide an abstraction for arbitrary UI frameworks, called *projects*. In this paper, we describe plain Squeak without its user interface, considering object collaboration, code execution, and extension points in the virtual machine. We implemented a command-line interface, the *Squeak Shell*, to emphasize the simplicity of adding a new UI to Squeak using this project concept. We believe that self-supporting programming environments can benefit from multiple user interfaces to accommodate a variety of tasks.

CCS Concepts

•**Software and its engineering → Integrated and visual development environments;** *Software libraries and repositories; Software prototyping; Object-oriented development;*

Keywords

User interface frameworks, Smalltalk, Squeak, bootstrapping, prototyping, direct manipulation, tool building, live programming, Morphic, model-view-controller, command-line interface, error recovery

PX/16, July 18 2016, Rome, Italy

 ISBN 978-1-4503-4776-1/16/07…$15.00
DOI: http://dx.doi.org/10.1145/2984380.2984386

1. INTRODUCTION

It is common practice to *build* new technologies with the existing technologies and eventually *replace* them. This practice is often referred to as *bootstrapping*. In the field of software engineering and research, bootstrapping is important for implementing new programming paradigms, languages, tools, and environments. For example, textual languages yield visual languages, keyboard input yields support for mouse or touch input, or text interfaces yield graphical interfaces. These new versions encompass not only novel ideas but also minor increments.

The bootstrapping process is essential for improving *live programming systems* [26]. Such systems have a long history and examples include Lisp (1958) [20], Smalltalk-80 (1980) [11, 10], Self (1987) [27], and The Lively Kernel (2008) [14]. In these systems, many parts can evolve such as the programming language, the runtime environment,[1] and the standard library shared for common programming tasks. In this paper, we focus on the evolution of the *user interface*, which is arguably a vital part considering the self-sustainability and liveness. In the beginning, there are usually external tools involved for writing code or debugging. However, the goal of such self-supporting systems is to modify applications and tools in use. For that, the edit-compile-run cycle is expected to be short to foster directness and liveness. For object-oriented systems, it is usually at the level of methods [11] or scripts [5]. Once there is an interactive user interface, programmers can start improving programming tools. The system can then evolve from within.

The user interface (UI) of Smalltalk-80 systems, developed at Xerox PARC, is called *model-view-controller* (MVC). The idea was to modularize source code for data, graphical representations, and user input to support reuse and extensibility. Squeak[2] – the Smalltalk system we focus on in this paper –

[1] Creating and maintaining virtual machines for interpreted languages is out of this paper's scope. There is, however, a strongly related case for the Squeak/Smalltalk virtual machine, which can be written in a subset of Smalltalk [13] and only a thin, platform-specific part in C.

[2] The Squeak Programming System, http://www.squeak.org

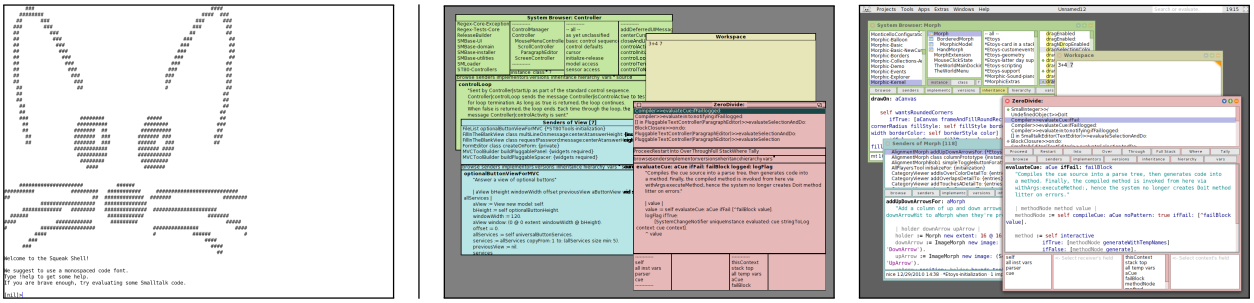


Figure 1: All three user interfaces (fltr: Squeak Shell, MVC, Morphic), which run in recent versions of Squeak. Each UI can be activated and used simultaneously according to the programming task. Usually, each UI has a custom application model and hence a custom set of interactive applications.

inherited and uses MVC in its first versions. Early achievements included a port of *Morphic* from Self to Squeak [17], which entails the idea of direct manipulation and a tangible user interface – properties that MVC is lacking. The process of introducing Morphic to Squeak led eventually to the point where MVC was not needed anymore to improve Morphic. From then on, Morphic tools dominated all programming activities. At the source code level, however, modularity was suffering. Two fundamentally different user interfaces entailed source code fragments that were scattered and tangled across the system.

Almost 20 years after the introduction of Morphic, the question about a shared abstraction for "arbitrary" user interfaces arises. There has always been the idea of managing running applications and user-specific content in a hierarchy of containers, called *projects*. MVC projects can embed Morphic sub-projects and vice versa. There is one top-level project that represents the primary user interface. Projects govern the basic use of processes as unit of code execution. Thus, UI frameworks build on top of the project concept and use existing primitives to handle user input and produce graphical output. We believe that this abstraction has the potential to form the programming interface for many kinds of user interfaces. The question is:

> What are the means and limitations to implement and bootstrap user interfaces in self-supporting programming systems such as Squeak/Smalltalk?

While one can add a layer on top of an existing UI [25], it might be beneficial to rebuild all means from ground up. The existing programming model might lack adaptability, additional dependencies might increase the maintenance overhead, or platform resources might dictate performance constraints. From a general research and prototyping perspective, we believe it is valuable to assess the UI bootstrapping capabilities of live programming systems such as Squeak/Smalltalk. Here, the overall goal is to keep on using the convenient tools from the existing UI for code writing or debugging. We want to avoid a freeze and lock-out of the live system.

We want to explore and discuss the existing facilities in Squeak that support bootstrapping user interfaces in general. For this, we implemented *The Squeak Shell*, which is a text-buffer-based, command-line interface with an interactive prompt and support for a simple application model. This approach resembles some characteristics of Bash, Emacs, Vim, and other text-based environments. The Squeak Shell poses an interesting contrast to the existing graphical interfaces in Squeak (Figure 1).

In this paper, we make the following contributions:

- Documentation of important milestones during the evolution of Squeak's user interface since 1996
- Description of Squeak's current core system as means to bootstrap new user interfaces
- Design and implementation of a Squeak Shell to illustrate a possible approach using Morphic's programming tools

All details about the current Squeak refer to the trunk build 16061 as of June 2016, which can be downloaded via http://www.squeak.org.

> *We believe that, among many other capabilities, Squeak is a valuable research and prototyping platform for various kinds of user interfaces, interaction models, programming paradigms, and educational methods.*

Section 2 provides more details about the historical evolution of user interfaces in Squeak. Then, section 3 describes important details about the system out of the context of user interfaces, including Squeak's standard library, flexible Smalltalk code execution, and the role of the virtual machine. In section 4, we explain how to implement a shell-like UI in Squeak and discuss bootstrapping. We conclude our thoughts in section 6.

2. THE HISTORY OF SQUEAK'S USER INTERFACE

The history of Squeak includes several milestones. While the evolution of the virtual machine (i.e. platform support) and its image format are very interesting in themselves, we focus on the graphical interface in this paper. Many prior

Squeak versions are still available online.[3] We explored these sources and collected important milestones related to the evolution of Squeak's user interface.

After a brief description of the two existing user interfaces, namely Model-View-Controller and Morphic, we show a timeline with screenshots to briefly visualize the state of the interface at that time.

2.1 Model-View-Controller

The MVC framework was introduced with Smalltalk-80 [4]. It separates code for handling data (model), user input (controller), and graphical representation (view). Typically, there are interdependent pairs of controller and view. By employing an observer pattern [9], models are decoupled from views and controllers and hence can be reused across the system. There is an extensive documentation with examples for MVC in "Inside Smalltalk - Volume II" by LaRonde et al. [16]

Interestingly, there are no convenient means to schedule (periodic) activities other than forking processes, which requires the programmer to take care of using appropriate locking and synchronization for data structures. Only the window with the current input focus executes code; other windows freeze by default until re-activated.

There are more recent frameworks for graphical user interfaces, which are inspired by the traditional MVC, such as model-view-presenter[4] and Qt's model/view architecture.[5]

The UI framework in Smalltalk-76 and -78 was also based on a desktop metaphor with windows. Every graphical object was a "window," even a star-shaped image. Nevertheless, MVC was especially designed for Smalltalk-80 to represent an extensible UI framework for many applications outside the research context.

2.2 Morphic "Version 2"

Morphic is an approach to introduce directness and liveness to a live programming environment. Direct manipulation interfaces [12] are considered to be user-friendly and efficient by visualizing objects and tasks interactively on screen. These interfaces are typically compared to command-line interfaces.

The first implementation of Morphic was for the second user interface in the Self environment [19]. The second implementation was for Squeak to replace MVC [17]. The third implementation was in JavaScript for the LivelyKernel [14].

In Squeak's Morphic, there is one UI process, which carries out the following activities in an endless loop: (1) query and process keyboard and mouse events, (2) run scheduled activities, (3) update display screen. Morphic is hence an implementation of the traditional "main loop with event handlers" pattern, which simplifies UI programming by executing all code in a single thread.

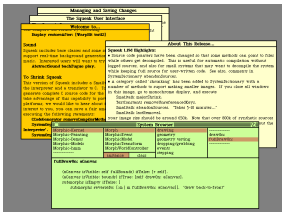
[3] http://files.squeak.org/, accessed on Aug 4, 2016
[4] http://martinfowler.com/eaaDev/uiArchs.html, accessed on Mar 30, 2016
[5] http://doc.qt.io/qt-5.6/model-view-programming.html, accessed on Mar 30, 2016

2.3 Timeline

In addition to the brief overview, we present the number of classes, number of methods, and lines of code as a measure of complexity and effort attributed to the user interface.

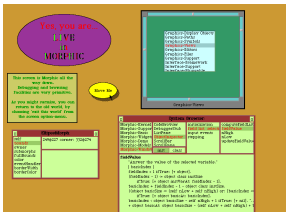
1996, Squeak 1.0 [6] Using Apple's Smalltalk-80, a new interpreter is written written in a subset of Smalltalk and translated into C code.[7] [13] Only a thin platform-specific layer needs to be written and maintained in C. The Smalltalk-80 library represents the code base for Squeak, including MVC as the graphical user interface. Soon after the initial bootstrapping process, subsequent versions of the interpreter were developed from within Squeak. The idea of *projects* as means of managing system content could be retained from Smalltalk-80 [16]. — Squeak's pioneers were eager to implement Self's Morphic [19, 27] and Fabrik [15] as an improvement over MVC to provide a direct manipulation user interface.

1997, Squeak 1.19 First appearance of Morphic-related code for demonstration, as such the first signs of what will become the second incarnation of Morphic.



MVC	107	classes	1 520	methods	12 454	LOC
Morphic	53	classes	981	methods	7 234	LOC
Other	210	classes	5 769	methods	60 750	LOC

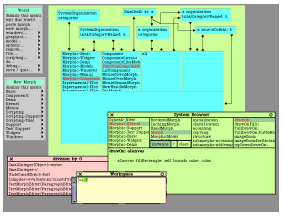
1998, Squeak 1.31 It is possible to open a Morphic world in an MVC view (window) to play around with morphs and the *halo* concept. A rudimentary full-screen variant is available to illustrate the use of morphs "all the way down." However, Morphic tools are very primitive and suggest strong dependencies on MVC.



MVC	109	classes	1 689	methods	13 666	LOC
Morphic	125	classes	2 231	methods	16 993	LOC
Other	262	classes	7 727	methods	82 741	LOC

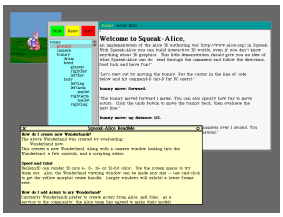
[6] The initial version of Squeak is not available. Our information is based on *Back to the Future: The Story of Squeak* [13].
[7] The interpreter of Apple's Smalltalk-80 was written in 68020 assembly.

1998, Squeak 2.0 The concept of projects was extended to support Morphic, which means that a full-screen Morphic project (resp. world) can be started to hide the MVC interface from the user. There can be projects within projects, worlds within worlds. Regarding a modular system structure, there is the idea of discarding all source code related to MVC or Morphic. However, that was not working yet. Then, there is the idea of *pluggable* views and morphs, which means to have configurable callbacks (e.g. `getTextSelector`) involving message sends via `#perform:`. These means of reusing models for both MVC and Morphic, however, entailed hard-wired construction methods in model classes such as `Browser` and `Debugger`.



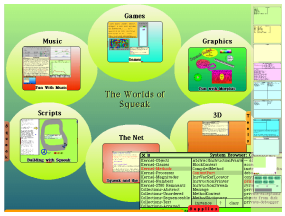
MVC	65	classes	1 463	methods	12 983 LOC
Morphic	157	classes	3 544	methods	26 222 LOC
Other	287	classes	8 385	methods	89 511 LOC

1999, Squeak 2.5 A Morphic project can be used as the top-level project. Discarding all MVC sources works now. Projects can also be exported to the file system to share not only code but also the whole workspace, filled with open tools and other objects.



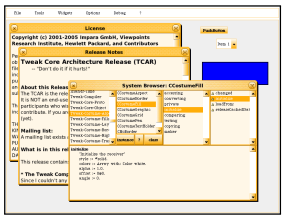
MVC	115	classes	2 610	methods	23 149 LOC
Morphic	218	classes	5 162	methods	39 334 LOC
Other	674	classes	14 284	methods	152 699 LOC

2001, Squeak 3.0 Images start with a Morphic project out-of-the-box. This means that Morphic is presented as the primary GUI framework for application developers. In contrast to MVC, there are many examples included to illustrate the flexibility of Morphic. Projects were used to structure these examples.



MVC	52	classes	854	methods	7 404 LOC
Morphic	395	classes	9 270	methods	71 751 LOC
Tools	66	classes	1 766	methods	17 809 LOC
Other	997	classes	22 008	methods	232 472 LOC

2004, *meanwhile* The idea of *Tweak* is born, which is a new viewing architecture and asynchronous events framework. It claims to combine ideas from Morphic and MVC.[8] While it has been successfully used in the Croquet project [23], development stopped in 2012.[9]

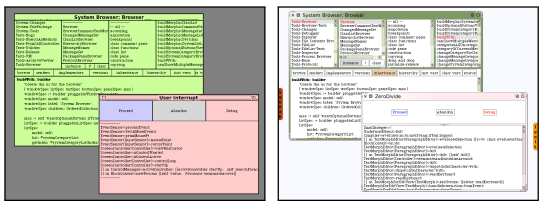


Tweak	565	classes	10 252	methods	70 638 LOC
Croquet	148	classes	3 236	methods	26 101 LOC

(Source code of Croquet 1.0.25)

2005, *unpublished* The idea of *Tool Builder* was born.[10] Major parts of the code for system tools, such as code browser and debugger, should be reusable for both Morphic and MVC. This separation cleans up model code and improves the modular architecture. Still, Tool Builder is not part of the contemporary release of Squeak 3.8.

2008, Squeak 3.9 The Tool Builder released and used. There is a builder [9] for Morphic and a builder for MVC. Many system tools were updated such as code browser, change sorter, debugger, process browser, and test runner. – There are many Morphic examples, which got separated into the category "MorphicExtras" thus reducing the Morphic footprint.



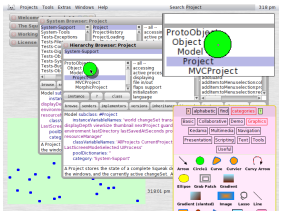
MVC	67	classes	1 295	methods	10 709 LOC
Morphic	184	classes	7 001	methods	54 865 LOC
Tools	58	classes	1 845	methods	19 619 LOC
Other	1 663	classes	34 760	methods	289 046 LOC

[8] http://web.archive.org/web/20110728001441/http://tweakproject.org/

[9] http://www.squeaksource.com/TweakCore.html, accessed on Feb 16, 2016

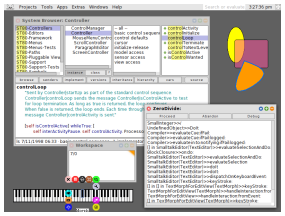
[10] http://www.squeaksource.com/ToolBuilder.html, accessed on Feb 16, 2016

2010, Squeak 4.1 There are now two kinds of projects for Morphic and MVC instead of tailoring the project concept to Morphic only. Regarding modularity, this sets the course for more project kinds in the future. The fully modular MVC can be removed and reloaded.



MVC	64	classes	1 447	methods	12 789 LOC
Morphic	186	classes	7 233	methods	57 809 LOC
Tools	61	classes	2 213	methods	20 315 LOC
Other	1 707	classes	34 451	methods	285 419 LOC

2015, Squeak 5.0 There were many improvements towards a modular system structure, which includes untangling MVC-specific and Morphic-specific code.



MVC	67	classes	1 563	methods	13 212 LOC
Morphic	204	classes	7 952	methods	61 682 LOC
Tools	63	classes	2 492	methods	22 879 LOC
Other	1 929	classes	38 955	methods	317 805 LOC

With the addition of Morphic to Squeak, the community recognized and separated common and specialized modules. This includes moving code from the MVC part to the core system. Then, the raise of Morphic entailed many examples, which obfuscated Morphic's core. It is an ongoing process to clean-up the system and improve the modular structure.

3. SQUEAK WITHOUT ITS USER INTERFACE

When the task is to implement a new user interface for a programming system like Squeak, one has to become familiar with means of everything besides such an interface. For one thing, there is the Smalltalk programming language and its semantics. Any program consists of objects that exchange messages to collaborate, hopefully producing the intended behavior. Application domains cover multimedia, games, simulation, and many more. Here is an example of the Smalltalk syntax:[11]

```
exampleWithNumber: x
  "A method that illustrates every part of Smalltalk method
  syntax except primitives. It has unary, binary, and keyboard
  messages, declares arguments and temporaries, accesses
  a global variable (but not an instance variable), uses literals
  (array, character, symbol, string, integer, float), uses the
  pseudo variables true, false, nil, self, and super, and has
  sequence, assignment, return and cascade. It has both zero
  argument and one argument blocks."
  | y |
  true & false not & (nil isNil) ifFalse: [self
  halt].
  y := self size + super size.
  #($a #a "a" 1 1.0) do: [:each |
    Transcript
      show: (each class name);
      show: '␣'].
  ^ x < y
```

This source code still compiles and runs in recent Squeak releases. In the following, we take a closer look at Squeak's standard library, its code execution model, and platform-specific functionality in the virtual machine.

3.1 The Standard Library

Everything is an object: numbers, files, sounds, or even classes. Squeak applications can build on a solid foundation of modules, which exhibit the Smalltalk language with its objects-and-messages metaphor in many domains. Many parts of the library are still in harmony with the original documentation of Smalltalk-80 [11], especially the topics about compilation, graphics, and collections.

Classes are organized in system categories. Each category has a common prefix to document and structure functionality:

Kernel-* Code compilation, class (hierarchy) (re-)definition, basic exception handling, process scheduling and synchronization, user input events, primitive types such as numbers and boolean values.

Chronology-* Everything related to time, time spans, durations, time stamps, etc.

Collections-* Everything related to working with multiple objects such as arrays, strings, streams, and dictionaries.

Graphics-* Everything related to graphical output, includes support for font rendering, drawing rectangles, and processing various image encodings such as PNG, JPG, and GIF.

Files-* Accessing the local file system using streams. Listing and navigating folders.

Network-* Using sockets to fetch network resources. Includes support for UDP, TCP, HTTP,[12] and HTTPS.[13]

Sound-* Managing audible output and reading various sound file formats. Includes support for FM, WAVE, and MIDI.

System-* Code change notification, object events, weak arrays and finalization, object serialization, concept of *projects*.

[11] Smalltalk Syntax on a Postcard:
http://c2.com/cgi/wiki?SmalltalkSyntaxInaPostcard

[12] WebClient, which is now part of the Squeak trunk:
http://ss3.gemstone.com/ss/WebClient.html

[13] SqueakSSL
https://github.com/squeak-smalltalk/squeakssl

Smalltalk has a powerful collection interface, which supports working with lists of arbitrary objects. The central structure is a fixed-size `Array` on which other kinds of collections build. For example, `OrderedCollection` is variably sized by managing growth with an internal array. `Dictionary` hashes its contents to indexes internally used to access an array. There is a pattern to convert between kinds of collections: "asKind". Examples include `asOrderedCollection`, `asSet`, and `asArray`. Having this, a temporary set representation can be used to remove duplicates. Lists can easily expressed as literal arrays (`#()`) or object arrays (`{}`):

```
"Filter odd numbers out."
#(1 2 3 4) select: [:ea | ea even].
"Map character to ASCII value."
{$a. $b. $c} collect: [:ea | ea asInteger]
"Iterate."
#(tree flower house) do: [:ea | "..." ].
```

The collection interface includes support for *streams*. At the platform level, contents from file or socket are usually processed as streams, which may be infinitely big. At the object level, streams work on top of collections to provide contents, which have a specific size.[14] For example, the construction of longer strings benefit from the stream interface in terms of memory usage; while "`('Hello', 'Again', 'World')`" involves five instances of `String`, streams can reduce it to four instances:[15]

```
String streamContents: [:stream |
                    stream nextPutAll: 'Hello';
                           nextPutAll: 'Again';
                           nextPutAll: 'World'].
```

Finally, there are *generators* to complement collections and streams. When iterating over a *computed* list of objects, there is no need to create that list in the first place. A common pattern is to provide a block, which represents the processing step, to the object that performs the computation. If this processing step cannot be accomplished at once, generators help by providing a stream interface for the content creation. Hence, there is still no need to create a temporary collection:

```
"Example taken from Generator class comment."
| generator |
generator := Generator on: [:g |
                Integer
                   primesUpTo: 100
                   do: [:prime | g yield: prime]].
[generator atEnd]
   whileFalse: [Transcript show: generator next].
```

Here, the generator yields prime numbers up to 100, which will be printed on the *transcript*. Traditional "printf()-debugging" or logging can be accomplished by writing strings, numbers, or boolean values on the transcript. For complex objects, programmers have to define a descriptive string representation. In Squeak's Morphic, there is a tool called "Transcript", which displays the latest output in a window. If the UI is currently not able to display such information, that output can be redirected to *stdout* or *stderr*. Any Linux shell or the Windows command-line prompt will then perform the display.

Object communication can be decoupled with several existing *observer patterns* [9]. First, there is the *change/update* mechanism, heavily used in MVC to decouple models from views. Objects have a list of *dependents*, who get notified on each "`self changed: #reason`" by calling "`dependent update: #reason`" synchronously. Then, there is a newer *object events system*, which allows for arbitrary callbacks to adapt to an objects interface. The dispatch can happen once at connection time and not repeatedly at notification time: "`subject when: #reason send: #dueToReason to: observer`". Finally, there are *system change notifications*, which allow tools to react to source code modifications.

In addition to dynamic object behavior (resp. run-time), programmers can add rules that affect the compile-time using *method pragmas*. These annotations remain close to the source code level, but can easily be processed at run-time because methods are instances of `CompiledMethod` and hence regular objects:

```
SoundService >> soundEnabled
   <preference: 'Enable sound'
      category: 'media'
      description: 'If false, sound is disabled.'
      type: #Boolean>
   ^ SoundEnabled ifNil: [true]
```

Squeak uses such annotations, for example, to store preferences close to the affected parts in the code to support version control.[16] The pragma syntax resembles the regular keyword message pattern. Arguments can be all literals such as strings, symbols, boolean values, and numbers.

Having this, user interfaces can provide flexible object communication protocols on top of the existing ones. There are objects "all the way down" to low-level tasks such as file access, there are data structures to work with collections of objects, and there are observer patterns to coordinate object-oriented behaviors.

3.2 Managing Code Execution in Squeak

User interface frameworks manage code execution to a certain degree by specifying the rules of querying input events, drawing to the screen, and accessing data. In general, frameworks employ the Hollywood Principle: "Don't call us, we'll call you." Application programmers can write source code without bothering about specific execution semantics. In Squeak/Smalltalk, objects collaborate via message sends, which are carried out in *processes*[17] as the unit of code execution and scheduling. The way processes are used influences the programming model, the impact of (programming) errors, and the overall responsiveness when facing long running computations or I/O lags (such as network requests).

[14] Unless repeatedly modified by another process.

[15] $2n-1$ compared with $n+1$, ignoring the collection's growth behavior

[16] Squeak's version control does not log objects in general but only source code and class definitions.

[17] In the domain of operating systems, the unit of execution is called *thread* and processes represent data structures to manage memory, file handles, or access privileges. In Squeak, threads are called processes.

Squeak's virtual machine provides *green threads*, which share one thread at the level of the operating system and CPU. While this can be a disadvantage for computation-intensive operations, it still promotes the possibility of designing responsive user interfaces. By default, Squeak's processes schedule in a cooperative fashion. When the current process yields, the next scheduled process becomes active and the current one goes at the end of the line:

```
Processor yield. "Next process at same priority will run."
```

Additionally, there are *priorities* and support for *interruption* by means of *semaphores* and other low-level synchronization objects. For each priority, there is a list of runnable processes. A process with a higher priority can interrupt a running process with a lower priority. An example of triggering an interruption is the use of delays, which is implemented using semaphores:

```
(Delay forSeconds: 2) wait. "Another process can run."
```

Here after 2 seconds, the process that executed that code becomes active again unless a higher-priority process is running or another process at the same priority is running. In the latter case, one will have to wait for that process to yield. Squeak defines the following process priorities by name:

- Timing Priority (80)
- High I/O Priority (70)
- Low I/O Priority (60)
- User Interrupt Priority (50)
- User Scheduling Priority (40)
- User Background Priority (30)
- System Background Priority (20)

There are usually 80 priority levels but this can be changed if no running or waiting processes are affected. User interface processes are expected to schedule at 40. This holds for the Morphic UI process and all MVC controller processes. Without the user interface, Squeak has the following processes running:

Timer Interrupt Watcher (80) Handles timing events such as delays in processes.

Low Space Watcher (60) Raises a warning if there is not much free memory left. Should prevent VM crashes. Process usually sleeps until triggered by the VM using a special semaphore.

Event Tickler (60) If no other process fetches user input events frequently, do it in this process. Required for user-controlled process interruption via the key combination `[CMD]+[.]`

User Interrupt Watcher (60) Waits for a special semaphore, signaled by the VM, that indicates user-controlled process interruption via `[CMD]+[.]` Works for processes below priority 60.

WeakArray Finalization Process (50) Waits for a special semaphore, signaled by the VM, to call finalization routines for registered weak structures after garbage collection.

Idle Process (10) Relinquish CPU time to the operating system.

Other process scheduling algorithms can be implemented as a high-priority, delay-based process or as a low-priority, non-preemptive process. Tweak and Croquet followed the latter approach. Any custom process (scheduler) can ask Squeak's scheduler about and fiddle around with all (runnable) processes according to any desired set of rules. Such rules known from other systems include quantum slicing, priority boosting, or putting a runnable process to the front of its queue. If there is additional information required – such as run duration so far and time of last file I/O – one can intercept the respective message sends to collect and provide that information manually. Processes are regular Smalltalk objects, whose classes can be specialized and new state or behavior be added:

```
(MySpecialProcess
   forContext: [ "Some computation ..." ] asContext
   priority: 40)
      resume.
```

Here is a very simple example of an additional scheduling process that runs at priority 70 and wakes up every 100 milliseconds to check whether the current process at priority 40 had changed or is still the same. If so, it will be re-scheduled to give the next process at 40 a chance to run:

```
[[ | list current previous |
   (Delay forMilliseconds: 100) wait.
   list := Processor waitingProcessesAt: 40.
   current := list ifEmpty: [nil]
                  ifNotEmpty: [:l | l first].
   (current notNil and: [current == previous])
      ifTrue: [list removeFirst;
                    addLast: current].
   previous := current.
] repeat ] forkAt: 70.
```

Note that processes will never be preempted by processes that run at lower priorities. With this additional time slicing rule, one can safely fork a process at the usual UI priority (40) while the system stays responsive:

```
[ [ "Some heavy computation ..." ] repeat ] forkAt: 40.
```

Otherwise, the user would have to interrupt the process manually by pressing `[CMD]+[.]` to make the user interface responsive again. Of course, the user can always execute such source code at lower priorities. If feasible, the computation could also yield its process frequently.

Programming mistakes can happen. If an object does not understand a message, the VM will call `#doesNotUnderstand:` on that object, providing an objectified version of the message send with all its arguments. By default, this situation is mapped to Squeak's *exception handling mechanism* by raising a `MessageNotUnderstood` exception. There are other cases where exceptions can occur such as `ZeroDivide`, `EndOfStream`, `KeyNotFound`, and `NonBooleanReceiver`. Exceptions can be handled by executing the suspicious code in a block:

```
[ 7/0. 3+3 ]
   on: ZeroDivide
   do: [:ex | ex resume].
```

Without `resume`, that expression above would return `nil` and not 6. If not handled by user code, the control flow will be redirected to the *debugger* where the buggy process will be suspended. The UI framework may has to spawn a new process to keep itself responsive to user input. In Morphic, for example, there is only one UI process, which has to be restarted if exceptions occur in it. In MVC, every window creates its own process when activated and hence debugger windows will not have to consider where the buggy process originates.[18]

Very long or (accidentally) endless loops can render the user interface unresponsive. If these computations are carried out below a certain priority level (usually 60), users can interrupt them by hitting `[CMD]+[.]`. Such interruptions will spawn debuggers where users can easily resume the respective computation. There is no automatic detection of endless recursion or loops, although one could easily implement one with a high-priority process as described above.

Most of Squeak's data structures are not thread-safe (resp. "process-safe"). Applications can use objects for access protection such as `Semaphore`, `Monitor`, and `Mutex` – which behave as commonly expected. There is a shared data structure called `SharedQueue`, which is protected by a semaphore. Squeak's Morphic, for example, uses shared queues to inject code snippets into the main loop or to query objects representing user input events. The scarcity of protection against concurrent access is common in comparable frameworks because precautionary checks increase maintenance effort and add performance overhead – many applications do not need that anyway. Regarding consequences of missing access protection, the VM will usually[19] not suspend but raise exceptions such as `MessageNotUnderstood`, `NonBooleanReceiver`, and `BlockCannotReturn` – which may appear arbitrary to the programmer but remain "debuggable."

Thread-safe operations include:

- Creating instances of classes
- Creating and scheduling process delays
- Creating, resuming, suspending, terminating processes
- Working with weak references (resp. weak dependents and weak registries)
- Fetching and processing user input events
- Any use of access protection objects and such that make use of them like `SharedQueue` does

Programmers can insert *breakpoints* into the code by raising a dedicated exception called `Halt`. The usual rules of exception handling apply here: if not handled, a debugger will be raised. Although programmers have to modify source code to add such breakpoints, they do not have to learn a new concept:

```
[self halt]
  on: Halt
  do: [:ex | "No breakpoints, please."].
```

Having this, user interface frameworks have many possibilities to schedule, execute, and debug Smalltalk code. Traditional patterns, such as one main loop plus event handlers, can easily be implemented. Also advanced script schedulers, such as found in Tweak, benefit from the simple but powerful, object-oriented abstractions in terms of processes, message sends, and method contexts (resp. stack frames).

3.3 Talking to the Squeak VM

Eventually, UI frameworks have to perform I/O operations such as reading keyboard input and drawing on screen. The platform-independence of Smalltalk code results from platform-specific VMs, which map and expose low-level functionality provided by the operating system to high-level Smalltalk objects. Graphical output, for example, is managed by a Smalltalk object that represents video memory. All pixel-based drawing operations are mapped to system resources, such as the DirectX protocol in Windows or the X11 protocol in Linux.

Every Smalltalk program makes *primitive calls* into the VM: either when performing platform-specific I/O or for improving code execution speed with a piece of specialized C code [6][13][7]. For example, there is VM support to speed up index wrapping when accessing an `Array`:

```
atWrap: index
   "Optimized to go through the primitive if possible"
   <primitive: 60>
   ^ self at: index - 1 \\ self size + 1
```

If the primitive call fails, the fall-back code will be executed as shown above. Otherwise, that code is just a readable explanation of what should happen in the VM. Primitive calls can also address *plugins*, which represent an important extension point in the VM. For example, all drawing operations end up in the `BitBltPlugin`:

```
copyBitsAgain
   "Primitive. See BitBlt copyBits, also a Primitive. Essential.
   See Object documentation whatIsAPrimitive."
   <primitive: 'primitiveCopyBits'
    module: 'BitBltPlugin'>
   self primitiveFailed.
```

Plugins can also be written in Smalltalk[20] to be compiled into C. They do not have to capture platform-specific behavior but can just contribute to the VM's modular architecture, like the `BitBltPlugin` does. If not critical to performance or security, it is common practice to implement as much functionality in Smalltalk as possible and only little C code to foster cross-platform support. Other examples for plugins include:

FilePlugin Accessing files and folders. Squeak uses it to write code changes and the object memory to disk.

[18] When resuming processes in MVC, debuggers have to activate the corresponding controller object if any. So, debuggers have to keep a reference to that controller.

[19] It is possible to manipulate special objects in certain ways, which disagree with the assumptions about regular Smalltalk code. Examples include "`SmallInteger removeSelectorSilently: #`>" and "`true become: false`". Then the virtual machine can crash or suspend.

[20] The subset of Smalltalk that can actually be translated is commonly called "Slang".

SocketPlugin Accessing local and remote network resources. Squeak uses it to receive code updates.

JoystickTabletPlugin Additional input devices such as joysticks and game pads.

SoundPlugin Synthesize sound in Squeak and play the samples on a sound card.

MIDIPlugin Process MIDI files in Squeak while talking to MIDI devices.

SqueakSSL Helps to establish secure connections to network resources. Used to establish HTTPS connections.

SqueakFFIPrims "Foreign function interface" (FFI) to use shared libraries that do not comply with the VM's plugin interface.

OSProcessPlugin Written in Smalltalk, provides access to OS inter-process communication such as pipes. Supports running shell commands and executing other OS processes. Thus, enables multi-programming by spawning more VMs and providing a communication protocol.

There are 819[21] methods that do primitive calls, from which 372 do not call into plugins but only the VM core. There are 51 168 methods in Squeak and this is hence only a small fraction of 0.016%. Most primitive calls can be found in `SmalltalkImage` (43), `Object` (24), `SmallFloat64` (20), `ContextPart` (19), and `SmallInteger` (19). `SmalltalkImage` uses primitives to read and write VM parameters, partially manage garbage collection, or snapshot the object memory to disk. `Object` provides a meta protocol to explicitly trigger message sends (i.e. `#perform:with:`, `#executeMethod:`), clone objects, access classes, or manipulate instance variables. `SmallInteger` uses primitives to speed-up arithmetic operations such as add, multiply, and compare.

In the context of primitive calls, UI frameworks have to fetch user input events frequently. In recent VMs, there is primitive 94, which fills an array with numeric values. These values encode the type – mouse or keyboard –, a time stamp, and type-specific payload such as screen coordinates for a mouse click. Then there is Smalltalk code to create event objects from this low-level representation. For example, the following array encodes a mouse event at roughly 22 minutes (1 334 728 milliseconds) after image start at screen coordinates (377, 328) where the left mouse button is hold:

```
#(1 1334728 377 328 4 0 1 0)
```

For Morphic applications, programmers can work with an instance of `MouseEvent` and check `#position` and `#redButtonPressed`. At the time of writing, however, such object-oriented event objects are specific to the Morphic framework.

Also in the context of primitive calls, graphical UI frameworks want to draw something on the screen while the screen should never flicker. Video memory is represented as a special `Form` instance, which is a two-dimensional, pixel-based surface to write on and read from. The primitive 102 will tell the VM about that object, which is also stored as a global variable in `Display` in Squeak. The primitive 106 will reveal the host window resolution and should be called frequently to detect resizing. The primitive 126 will tell the VM whether to forward all calls directly to the screen or to defer the updates until primitive 127 is called. From Squeak's perspective, this concept resembles *double buffering* but implementation details can vary between platforms.

There is usually one host window per VM in the operating system. This is no limitation in general but seems to be a historical design decision because the Smalltalk system could replace the whole operating system programming and user interface [4]. It is feasible to extend the VM with plugins to support multiple host windows.[22]

Having this, the VM's plugin architecture, as well as the FFI plugin itself, provide simple means to add advanced hardware support. Elaborate visualizations may benefit from hardware-accelerated graphics. Exotic input devices may provide additional parameters. UI frameworks can consider these possibilities in their design and balance the trade-offs with platform lock-in.

4. HOW TO CREATE THE SQUEAK SHELL

The focus of this paper is to illustrate the simplicity of adding a new UI framework to Squeak from within an existing framework – Morphic in this case. Therefore, we build on the details of the previous sections and explain how to add a *shell interface* to Squeak, which will be a third one next to Morphic and MVC. Instead of designing the Squeak Shell as, for example, a Morphic application, we want to explore Squeak's core facilities. Due to its simplicity, such a shell interface is a great example to highlight the existing UI patterns that many other UIs would also have to employ. These patterns include *project*, *user interface manager*, *application registry*, and *tool builder* as well as *interactive debugging*.

While creating a new UI for and from within Squeak, our goal is to take advantage of such a self-supporting system. We want to keep using the existing tools while working on their next generation. We want to avoid a lock-out and the domination of external tools. At first, our new UI will support code writing, running, and debugging. So, these first applications will be programming tools, which represent the foundation of an application model. Then, users are enabled to create new content. We evaluate and discuss bootstrapping at the end of this section.

4.1 Squeak's Common User Interface Patterns

Projects are entry points into user interface frameworks. When entering a project, it should spawn initial processes to query input events and draw the state of things on screen in an endless loop. Projects need to keep track of their vital processes to re-spawn them in case of an exception. There are *deferred UI messages*, which are usually objects stored in a `SharedQueue` to call `#value` on if there is enough time left in a cycle - improving responsiveness. Projects form a hierarchy and support users to organize their contents (Figure 2). At the

[21] Some methods call the same primitives. No quick returns.

[22] Project "Areithfa Ffenestri" http://wiki.squeak.org/squeak/3862, accessed on Mar 31, 2016

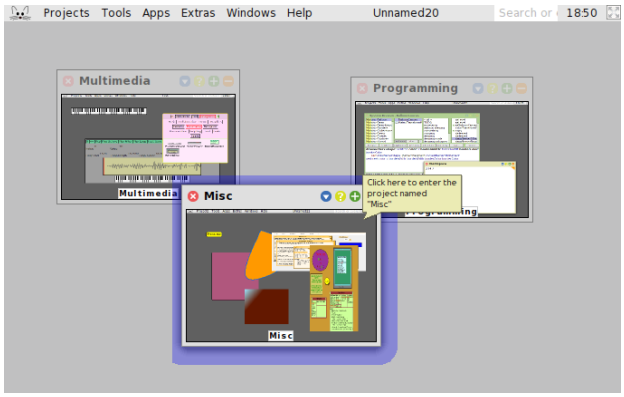
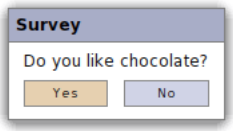


Figure 2: Morphic keeps track of the project hierarchy with project windows. Users can enter a project with a click on the window. Closing windows means discarding projects.

time of writing, background projects sleep and are requested to terminate their processes when entering another project.

Every project has a *User Interface Manager*, which dispatches common dialog-based tasks such as asking to confirm an operation, indicating progress, requesting text input, and choosing from a list of alternatives. In Morphic, the `MorphicUIManager` creates, configures, and opens specific morphs for that. The UI Manager is not meant to be modified by applications but used to provide a *consistent look-and-feel* for such common tasks. For example, to ask a yes/no-question, application code would call this:

```
answer := Project current uiManager
   confirm: 'Do␣you␣like␣chocolate?'
   title: 'Survey'.
```



Squeak has *Application Registries*, which represent a convenient interface for more complex tasks that can be carried out by various applications. There is the registry `MailSender` to send emails, `SoundService` to play sounds, `WebBrowser` to open Web sites, or `ToolSet` to perform programming tasks. Two similar applications would register and the user has to choose a default. Having this, Squeak is prepared for many kinds of activities, not just programming. Depending on the application model, new kinds of projects can register their own applications. In the case of Morphic and MVC, the tool set with all programming tools is shared by employing an additional builder pattern [9].

```
"Inspect the state of the current project."
ToolSet inspect: Project current.
```

Tool Builders construct tools based on specs. In Squeak, tools usually consist of one window with several lists, text

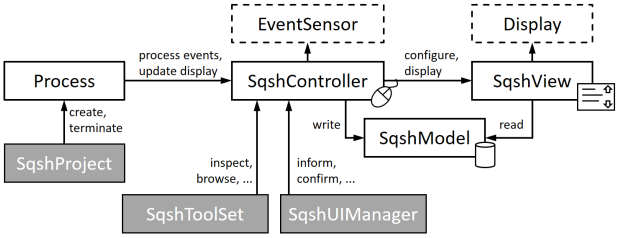


Figure 3: The Squeak Shell architecture. Gray boxes are singletons/globals, accessible from application code. Dashed boxes are I/O paths into the VM. Note that this is no actual MVC but a similar pattern.

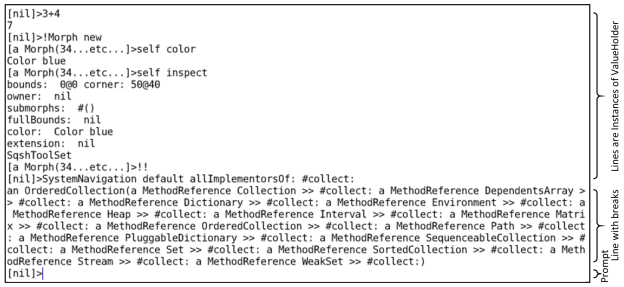


Figure 4: In the Squeak Shell, every line is an instance of `ValueHolder`, which supports the change/update observer pattern to notify the view of updates.

fields, and buttons. A tool's model class implements `#buildWith:`, which gets called with a concrete builder as argument. Builders provide specs and means to construct the real widgets. Specs store abstract information such as callbacks, tool tips, and layout hints. There are composite specs (windows or panels) managing children and normal widget specs (buttons, lists, etc.) being the children. Communication between the tool's model and the widgets is decoupled with the change/update observer pattern as described in the previous sections. Note that, for a new kind of project, it is not necessary to implement a tool builder because programming tasks first have to go through the current tool set, which is an application registry and can already be project-specific.

Given the global accessibility of projects, application registries, and tool builders, working on a new UI framework can only partially happen from within an existing UI framework. Programmers can try out only some aspects, such as a new main loop, in an other UI's process. There will be the point where a new project has to be entered and its process to be spawned – while the current process terminates. Then if not carefully thought through, lock-out of Squeak is possible. However, a locked image will not be stored on disk and hence programmers can safely resume from the last checkpoint.

4.2 Squeak Shell Architecture

The Squeak Shell has a controller, a model, and a view. We hence follow an MVC-like pattern and separate input processing, information storage, and text display as depicted in Figure 3. The *model* is, first of all, a buffer for text lines,

which are objects[23] whose contents are tracked and whose changes are immediately displayed. Then, the model stores the current prompt, previous commands, the text cursor position, and a list of variable bindings to support code execution. The *view* knows the model to show all lines in a list. It adds support for line wrapping if the screen is not wide enough as depicted in Figure 4. It is configurable with a background color and a default text font and it supports vertical scrolling. The *controller* dispatches user input, which is basically text input, cursor navigation, and command execution. It knows the model to, for example, update the prompt, and it knows the view to trigger redraw on screen.

If the user enters the expression "`3+4`" followed by `[Return]`, that expression will be printed on the prompt, the result will be printed on the next line, and an empty prompt will appear as a third line (see upper part in Figure 4). There are four key strokes, which result in two new lines in the buffer, an empty prompt to be filled again, and about four screen updates between. Besides the model triggering screen updates lazily, the shell frequently suspends its main process to ensure a minimal cycle duration of 20 milliseconds to save CPU time.

The Squeak Shell evaluates code synchronously. It works like a "do it" in a Smalltalk workspace known from MVC and Morphic. The model holds a dictionary with bindings, which are accessed through the controller. The controller is notified during compilation of any issues such as missing bindings. Here is how the controller evaluates code :

```
result := Compiler new
            evaluate: expression "from the prompt"
            in: self model context "temps, not used"
            to: self model receiver "resolve 'self'"
            notifying: self "provides more bindings"
            ifFail: [^ self] "parse/compile errors"
            logged: true. "log expression in changes"
```

After successfully parsing and compiling the expression, code evaluation can still raise an exception. There is no generic exception handler in the Squeak Shell. Instead, we attach to Squeak's debugger interface, which is used for all unhandled exceptions as described in the previous sections. Yet, the shell has no interactive debugging support but only displays the stack and presents an empty prompt.

In the shell, all programming tools are quite primitive. While transcript output and progress display work as expected, usually interactive tools produce only static text output. For example, the object inspector lists a string representation of all instance variables (see middle part in Figure 4). Nevertheless, there is only little code required to write and integrate such primitive UI managers and tool sets. Being able to evaluate code, users could change classes, compile methods, and evolve the system. Anyhow, this is far from being convenient.

The next steps include the elaboration of the application model. Similar to MVC, multiple controllers could exist side-by-side or be nested. For example, opening a text editor from the shell could be realized by adding a sub-controller to the shell controller. The tool set and the UI manager could be extend to provide interactive tools and dialogs. This simple design of the Squeak Shell is arguably not very restrictive and open for extension. There could also be a custom tool builder to reuse existing tool models, which are already shared among MVC and Morphic. The Squeak platform itself expected only the creation of three classes: `SqshProject`, `SqshToolSet`, and `SqshUIManager`.

[23] Squeak's class `ValueHolder` is a mini model that automatically notifies its dependents - here, the shell model - about changes.

4.3 Step by Step

Programmers should benefit from the existing UI framework as much as possible while creating a new one. This includes writing code, running code, testing and debugging code. Back in 1998 in Squeak 1.31, an early version of Morphic was rendered in an MVC view. Programmers could try out the new features without having to leave their familiar context. While it might be troublesome embedding two fundamentally different UI frameworks in each others "information containers," running the main loop inside another main loop might be sufficient to test the basics.

We suggest the following procedure to implement the Squeak Shell from within Morphic. The ⚠ denotes a strong probability to freeze or lock the image by accident:

1. Create and test classes for model and view.
2. Create controller and run event/draw loop on top of the Morphic main loop.
3. Handle *escape key* in controller to return to the previous (Morphic) project.
4. ⚠Spawn the shell's main loop in a custom project's process.
5. ⚠Implement debugger pattern; print the stack and restart the process if needed.
6. Improve programming tools, add applications.

While the steps 1, 2, 3, and 6 remain debuggable with Morphic tools, entering a custom project and interfacing the debugger mechanism might result in an image freeze (⚠). Morphic's debugging facilities will only help when the current project and tool set will invoke Morphic's project code to respawn the UI process to keep the UI responsive. If an unhandled exception ends up in an unimplemented interface and the current project raises the `MessageNotUnderstood` error, the exception mechanism will recursively call itself – lock-out. Squeak already has a mini shell for primitive error handling, which can revert the last code change, but project-specific code has to make use of it. Simplified, an implementation of the debugger pattern looks like this:

```
debug: process context: context message: string
  "Keep UI responsive by respawning process."
  Project current
     spawnNewProcessIfThisIsUI: process.
  "Schedule debugging for the next cycle."
  Project current addDeferredUIMessage: [
     [ "Do something with the buggy process."
        Project current uiManager
           inform: string;
           inform: context stack printString.
```

```
        process terminate.
    ] on: Error
      do: [:ex |
          "If the project's debugger fails..."
          self primitiveError: string]].
"This process will be handled as above."
process suspend.
```

This is a very important use case for deferred UI messages, which have to be processed in the project's loop. The example above just prints the error message and the current stack, then terminates the buggy process. Morphic and MVC open a window to support interactive debugging with "step over," "step through," "step into," and so on. During debugging, code execution will be *simulated* in Smalltalk. Suspended processes can be resumed at any time to be scheduled as usual by the VM. Such features can easily be added to the Squeak Shell.

Processes of UI frameworks should avoid *busy waiting* so that the VM can relinquish CPU time to the operating system. This can be achieved with a delay like Morphic and MVC do. This is also required to give other processes below UI priority a chance to run. If Squeak's idle process, which runs at priority 10, becomes active because all other processes are blocked, it will repeatedly tell the VM to give one millisecond back to the operating system.

All bugs introduced to the main loop, which cannot be resolved by restarting the main loop, will freeze the image. For drawing objects on screen, Morphic provides specific error handling that flags bad morphs to not be redrawn in the next cycle. Squeak has no mechanism to fall back to another project (kind) that might be capable of keeping the system responsive. However, this is no technical limitation but affects the current design of projects and error handling. One idea is to have a shortcut like `[CMD]+[.]` that does not only suspend the current process and restarts the UI loop, but a shortcut that enters any other runnable project from which the respective problem could be debugged. Another idea is to find a better place for the primitive error handling as described above. Such primitive error handling could first try to enter any other runnable project and only then spawn a rudimentary read-eval-print loop.

4.4 Recover From Mistakes

Squeak provides a great deal of flexibility and thus various ways to "shoot yourself in the foot" by accident. Examples include endless loops of message sends, infinite non-tail recursions that fill the stack, not returning VM plugin calls that block user input, and inadvertent termination of essential processes. Programmers can usually recover from all these pitfalls by saving and duplicating the image file frequently. Even if the time of the last snapshot was a while ago, there are several means to recover from within the image.

Squeak writes all source code changes and code evaluations ("do-its") into the `.changes` file, which is hence an ever growing log of all code-centric activities. A regular image snapshot will usually synchronize the `.image` file, which contains the whole object graph, with the `.changes` file in terms of source code pointers. Without that snapshot, the latest changes are unknown to the image in the sense that some methods do not point to the most recent versions of their sources. However, there is a simple way to recover unsaved changes with the *Change Recovery* tool, which scans the `.changes` file for these artifacts and supports programmers to load them again.

For endless loops or infinite recursions, there is no need to shutdown the VM. There is usually the possibility to hit `[CMD]+[.]` to interrupt the current process to debug it. For Morphic, this is usually the single UI process because most of the code is evaluated as an effect of user interaction such as mouse clicks. MVC and Morphic do both prevent the user from interrupting important processes by accident; they do restart the UI process if necessary.

If the available memory gets low, there will be an exception raised so that the programmer can decide whether to continue the computation at own risk or to revise the source code. The VM will not suspend or terminate suddenly before warning the user.

In Morphic, morphs will not be re-drawn if they did raise an exception once in their drawing code. The programmer will notice that in terms of a red rectangle with a yellow cross spanning the morph's bounds. That morph will still be in the world and accessible via its halo to be explored and debugged. In general, UI frameworks are free to implement any partial error handling to remain "debuggable".

In the end, programmers should be aware of the core mechanics in such a system to recover from errors. They should now about the `.changes` file, the way code execution works, and how `[CMD]+[.]` can work even if no regular event processing takes place in the image. This does not only affect builders of UI frameworks but all application programmers who work in Squeak or similar self-supporting environments. These means to recover can save a great amount of time and foster efficient working habits. There is no reason to be afraid of evaluating "`true become: false`" in your production image.

4.5 When to Bootstrap?

Since the first steps of Morphic into Squeak's MVC, inherited from Smalltalk-80, there has been effort to separate both UI frameworks and to support the unloading respective code. It took about seven years until 2001, when Morphic became the user interface of Squeak 3.0 with many tools, applications, and other multimedia content. Thus, we conclude that it is not only about the technical feasibility to bootstrap, to leave the old framework behind. People have to accept the different look-and-feel, learn the new features, and eventually adapt their working habits. For this transition, there has to be content. It cannot just be the new framework, but examples that show how to make use of it, to illustrate the benefits, to stimulate creativity.

Our Squeak Shell implements a command-line pattern, which is very familiar in the Linux community. It's simplicity supported our focus on Squeak's bootstrapping capabilities for new UI frameworks. By being able to evaluate any piece of Smalltalk code, programmers can modify the whole Squeak system from within the Squeak Shell. However, ad-

vanced programming facilities, such as method editors and class browsers and interactive debuggers, are likely to have a great impact on the productivity in such a shell. For reference, emacs and vim provide many other valuable extensions for this UI metaphor.

We do not think that the ultimate goal of separating Morphic and MVC is to discard one for another. In self-supporting, live programming systems such as Squeak, programmers can benefit from having several working user interfaces. If programmers can freely choose their UI, they can accommodate domain-specific tasks or address challenging debugging scenarios from different angles. There is no need for one UI to surpass another in every aspect.

5. DISCUSSION

After describing the history of Squeak's user interfaces, its technical means to build a new one, and the Squeak Shell as a practical example, we now take a step back and reflect on the general motivation, lessons learned for Squeak and other environments, opportunities for research and industry, and possible next steps for Squeak itself.

5.1 Three Different Goals

We described Squeak's two UI frameworks briefly and our Squeak Shell in more detail. While this paper focuses on Squeak's current boostrapping capabilities, the design goals for each of the three frameworks are quite different. Their chronological order should not be confused with their, basically orthogonal, intents.

When Smalltalk-80 was designed based on Smalltalk-72 and -76, one main driver was to create a usable Smalltalk environment, which has a chance to become popular outside its research context at that time. MVC was the first coherent application framework and all the programming tools were created with it. To the best of our knowledge, any bigger application such as the Alternate Reality Kit [24] and The Analyst[24] had to step beyond the capabilities of MVC to support richer graphics and interactions.

When Morphic was added to Squeak, there was already experience gathered from the existing implementation in Self [19]. Interactivity, direct manipulation, and multimedia composition: the creators of Morphic moved away from professional programmers towards novices including children and their very first contact with computers and programs. Comparing it with the MVC framework and its focus on programming tools, Morphic simplifies rich content authoring and sharing in general. Programming tools are basically just a special form of applications in this context. Other applications such as Alice [3], EToys [1, 8], and Scratch [18] could benefit from Morphic's concepts in a notable way.

While we do favor live programming systems with interactive and graphical content/tools, the decision to add a command-line interface to Squeak was merely driven by its simple implementation. MVC and Morphic are more complex because of their support for graphics-based content authoring, mouse input, and advanced event handling. The Squeak Shell is text-based and provides only minimal functionality to execute Smalltalk code to interact with Squeak's shared object space. This minimalism is not only beneficial for describing Squeak's bootstrapping capabilities. Moreover, its reduced complexity does entail a reduced maintenance effort and will hence be a robust safety net if the more complex frameworks fail to function. Our recent efforts to clean-up Squeak's project concept is already part of the main branch and the Squeak Shell will be soon, too.

[24]The Analyst was a document-based, multimedia authoring system with hyperlinks to connect pieces of information. It provided spreadsheets, charts, outlines, images, maps, and other kinds of interactive visualization.

5.2 Lessons Learned From Squeak's Projects

Projects represent the entry point for UI frameworks to start initial processes and govern code execution and basic interaction patterns between objects. Projects can also be used to organize content, such as windows and other graphical objects, in a hierarchy (Figure 2). In live programming environments, such a projects concept should be located directly above the base system with its standard library, means to run code, and all VM extensions. Any higher level application or framework should be a project to some extent. Although there will always be other processes running (see subsection 3.2), projects are meant to establish and maintain the "liveness" in such environments. Compared to "projects" in traditional programming environments such as Visual Studio or Eclipse, Squeak's projects do not only manage structure but also behavior.

We learned many interesting aspects of Squeak while implementing the command-line because we had to refactor many parts of the system to improve modularity and minimize the expected interface of such a new project kind. We realized that debugging is closely related to code execution and that the invocation of the debugger is "too far away" from projects, which are actually responsive for (re-)starting the main loop. At the time of writing, an unhandled exception invokes the debugger via the current tool set. Only then, the debugger considers the current project for interrupting, resuming, or terminating the UI process. In addition to such superfluous indirections, we stumbled upon several global variables used to control either MVC or Morphic. If we did not manage to shutdown and startup projects correctly when switching between them, it was likely to lock-up the system. For example, there is the method `SmalltalkImage >> #isMorphic`, which used to check the global `World` not being `nil`. This was one of many challenges posed by the predatory use of global state.

It turned out to be very beneficial to have a shared object space. While plugin architectures in environments such as Eclipse provide only a limited interface to access and manipulate data, Squeak reveals everything with classes-as-objects and a powerful meta-object protocol including `#allInstances`. Advanced Eclipse plugins such as CodeBubbles do have to make serious efforts to mold the user experience while reusing the underlying Eclipse platform [22]. There are several examples out there where plugins had to

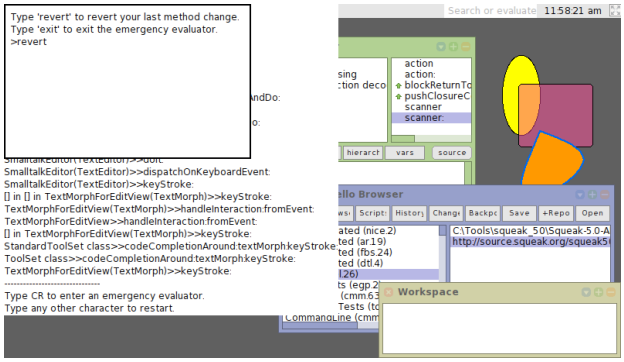


Figure 5: Squeak's emergency evaluator (top left) with basic functionality to run Smalltalk code to recover from a locked system state by, for example, reverting the previous modification.

generate the whole AST by themselves because Eclipse did not provide access to its own low-level structures. This is arguably no implication of having a file-based system — while Squeak is an image-based one — but rather of designing a system without some kind of meta-object protocol or query language. However, there are efforts made to mitigate this issue by providing scripting interfaces to such environments. This is a first step into the right direction.

In Squeak, there are no generic means to safely execute projects in background. There is a shared object space and every project assumes that it is alone in the system. Data structures are usually not thread-safe. The only solution would be to explicitly account for background activities so that projects do not only release computation power to the operating system (see subsection 3.2) but also to other projects. Such well-defined, cooperative context switching might enable concurrent project execution. In Morphic, such a well-defined switching point is in `WorldState >> #doOnceCycleFor:`.[25] In MVC, it is `Controller >> #controlLoop`. Additionally, Morphic provides higher-level means to make object interaction concurrent: stepping. Steps are like input event handlers, they have to complete quickly to not impair the responsiveness of the UI. Any object can step and morphs use it to implement animation. Stepped code is always executed in the UI process. In the general sense, however, projects can do what they want and employ additional process schedulers as described before. Any application can spawn new processes. Since Squeak's process forking mechanism is conceptually below the project concept, it is not in the respective framework's control. It remains tricky to let background projects tick.

Projects can be used to circumvent Squeak's emergency evaluator (Figure 5), which is usually the last hope to revert the system to a usable state in case of a serious problem. If we assume that different project kinds are implemented quite differently and hence share aspects from the base system only, entering another project kind can help debug serious issues of a UI framework. For example, if you introduce a bug in Morphic's list widgets, you cannot draw the debugger window, which uses lists to show the stack frames, and hence you will not be able to debug the problem with Morphic's tools. Then, the emergency evaluator would usually appear. Although there do not seem to be (m)any MVC programmers in the Squeak community, one would arguably be more happy to fix such an issue with more sophisticated MVC tools than with the limited emergency evaluator.

There might be that one convenient or efficient tool for a common problem scenario...but it did not get implemented for the new environment you are working in right now. Bad luck. However, Squeak's shared object space in combination with the projects can help to just launch that tool in the older environment and run it on the problem data. Such tools include analysis, visualization, or interactive code transformation. Considering programming tools, you can benefit from keeping old UI frameworks up and running – like MVC. This is strongly related to having a shared object space as mentioned above. Otherwise, programmers would have to find a common representation to be understood by both tools, like files with chunks of semi-structured text.

During all of our experiments with the Squeak Shell, the VM kept running. If programmers provoke a stack overflow, there will usually be an exception raised, which pauses the broken computation to let the user decide. If primitive calls (see subsection 3.3) fail, there are exceptions raised, too. However, if a failing primitive relates to frequent activities such as user input event handling and screen updates, then there is no chance to debug that. The environment will get stuck.

5.3 From Research Prototypes to Useful Applications

We believe that, among many other capabilities, Squeak is a valuable research and prototyping platform for various kinds of user interfaces, interaction models, programming paradigms, and educational methods. Squeak can also be used to create many kinds of useful applications, be it in Morphic, MVC, or on top of any other framework. The platform is very flexible and user experiences are moldable to accommodate any domain, task, or personal preference.

In this paper, we focus on bootstrapping UI frameworks, but the transition between applications and frameworks is fluent. Content authoring tools such as Etoys [1, 8] look like frameworks themselves, while they depend on other frameworks to provide tools for lower-level tasks such as code writing and debugging. There may be no need to bootstrap because the usage scenarios overlap only in parts. Not every user is a professional programmer, not every programmer can benefit from the means designed to help domain experts or children.

The Parks PDA [21] implements a hypercard-like interface on top of MVC. There is a single application with a main loop running on top of an MVC controller loop (compare to subsection 4.3). Any unhandled exception will reveal underlying programming tools. While many users could

[25] In Squeak's Morphic, there are some faint traces of doing a world cycle in background. This was, however, never fully implemented.

feel helpless in that situation, programmers could still fix and improve the live system conveniently. The Parks PDA interface also supported content artists to communicate their ideas to programmers, which could in turn keep using their well-known tools. No need to implement programing tools in that hypercard interface. No need to bootstrap.

Further examples include Vivide [25], which builds on top of Morphic and relies on the traditional debugger to some extent. There is Scratch [18], which originally used the Morphic in Squeak 2.8, that tries to catch all errors in scripts to help non-programmers as much as possible. The tODE[26] environment also appears as a single application in Morphic to support administration tasks in the GemStone object database management system [2].

Despite non-existent examples, you could employ the concept of projects to write "native" Squeak/Smalltalk applications. While the VM would still be running as a regular process in your operating system – such as Windows, Linux, and Mac OS – the application code would not build upon a framework such as Morphic but it would have its own main loop and directly do primitive event handling, display drawing, and so on – like writing a game with SDL.[27] Squeak would take on the sole role of an execution environment with some libraries. For example, one could think of a Telnet server that exposes the object space to other systems. To some extent, this would be comparable with GNU Smalltalk,[28] where you typically[29] run Smalltalk scripts that were written in some external environment. When building such "native" Squeak applications, programmers would still write code with the tools in Morphic or MVC. Even interactive debugging would work from within another UI framework because Squeak offers sophisticated means to manipulate processes and stack frames. UI frameworks and their tools could remain pluggable, that is, loadable and unloadable as required. For application deployment, one could easily[30] strip off the tools. In case of a serious error, there is always the emergency evaluator (see Figure 5) where any Smalltalk code can be evaluated and hence some programming framework can be attached on-the-fly. Theoretically.

5.4 Next Steps

There are practical aspects that we plan to address to further improve the Squeak platform considering UI frameworks and "native" applications:

- Add the Squeak Shell and install it as fall-back project for serious errors in Morphic or any other project kind.
- Improve the means to debug one UI framework from within another by supporting "proceed" and hence resuming the project that failed to handle the exception by itself.
- Reduce the footprint of new project kinds by making them invoke the emergency evaluator (or a fall-back project) by default if unhandled exceptions occur. Further reduce the risk to lock-up the environment.
- Add `NativeApplication` as a subclass of `Project` to provide a simple interface for Smalltalk projects that do not intend to benefit from Morphic or MVC.

There are also research interests and open questions that we intend to pursue to find out more about the capabilities of Squeak:

- What are the actual limitations for user interface frameworks to be implemented in Squeak? Do VM extensions represent a feasible option for UI programmers or are those too challenging?
- Is there a theoretical model to describe the risk of locking up the environment? Can it be improved?
- Can you compare the implementation effort for an interface in Squeak with another (non-live) environment?

6. CONCLUSIONS

We presented milestones of Squeak's UI history. The initial implementation of Morphic happened from within MVC, then it was improved from within itself. Meanwhile, Squeak's *projects* evolved into an abstraction for other UI frameworks.

We described Squeak's means to work with objects, manage code execution, and invoke platform-specific functions in the VM. Based on this, we implemented a command-line interface to discuss and evaluate the project concept.

We think that user interfaces represent a vital part in self-supporting systems. Next to the virtual machine and the programming language, the user interface can always benefit from novel ideas, prototypes, and working implementations. Researchers and application developers should strongly consider realizing their ideas in such environments to benefit from immediacy, liveness, and directness.

Acknowledgments

Thanks to Eliot Miranda, David T. Lewis, Tim Rowledge, Bert Freudenberg, Jens Lincke, Dan Ingalls, and the entire Squeak community for answering our questions personally or on the mailing list. We owe deepest gratitude to Andreas Raab, who contributed to Squeak with ever so much diligence and effort. Sincere thanks also go to all PX workshop participants, who provided valuable feedback by discussing this topic thoroughly. We gratefully acknowledge the financial support of HPI's Research School[31] and the Hasso Plattner Design Thinking Research Program.[32]

[26] The Object-centric Development Environment: https://github.com/dalehenrich/tode, accessed on June 17, 2016

[27] Simple DirectMedia Layer: https://www.libsdl.org, accessed on June 16, 2016

[28] http://smalltalk.gnu.org, accessed on June 16, 2016

[29] There are GUI frameworks for GNU Smalltalk such as Blox: https://www.gnu.org/software/smalltalk/manual/html_node/Blox.html, accessed on June 16, 2016

[30] In Scratch [18] and Etoys [1, 8], the Morphic environment had to be sealed with some effort to hide programming tools. Other examples include OpenQwaq (github.com/itsmeront/openqwaq) and Impara's Plopp (www.planet-plopp.de).

[31] www.hpi.uni-potsdam.de/research_school

[32] www.hpi.de/en/research/design-thinking-research-program

7. REFERENCES

[1] B. J. Allen-Conn and K. Rose. Powerful Ideas in the Classroom. *Viewpoints Research Institute, Inc*, 2003.

[2] P. Butterworth, A. Otis, and J. Stein. The GemStone Object Database Management System. *Communications of the ACM*, 34(10):64–77, 1991.

[3] S. Cooper, W. Dann, and R. Pausch. Alice: A 3-D Tool for Introductory Programming Concepts. In *Journal of Computing Sciences in Colleges*, volume 15, pages 107–116, 2000.

[4] L. P. Deutsch. The Past, Present and Future of Smalltalk. In *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP)*, pages 73–87, 1989.

[5] T. Felgentreff, J. Lincke, R. Hirschfeld, and L. Thamsen. Lively Groups: Shared Behavior in a World of Objects Without Classes or Prototypes. In *Proceedings of the Workshop on Future Programming*, pages 15–22. ACM, 2015.

[6] T. Felgentreff, T. Pape, L. Wassermann, R. Hirschfeld, and C. F. Bolz. Towards Reducing the Need for Algorithmic Primitives in Dynamic Language VMs Through a Tracing JIT. In *Proceedings of the Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*. ACM, 2015.

[7] B. Freudenberg, D. H. Ingalls, T. Felgentreff, T. Pape, and R. Hirschfeld. SqueakJS: A Modern and Practical Smalltalk That Runs in Any Browser. In *Proceedings of the 10th ACM Symposium on Dynamic languages (DLS)*, pages 57–66. ACM, 2014.

[8] B. Freudenberg, Y. Ohshima, and S. Wallace. Etoys for One Laptop Per Child. In *2009 Seventh International Conference on Creating, Connecting and Collaborating Through Computing (C5)*, pages 57–64. IEEE, 2009.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Abstraction and Reuse of Object-oriented Design*. Springer, 2001.

[10] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley Longman Publishing Co., Inc., 1984.

[11] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[12] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct Manipulation Interfaces. *Human-Computer Interaction*, 1(4):311–338, 1985.

[13] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak—A Practical Smalltalk Written in Itself. *ACM SIGPLAN Notices*, 32(10):318–326, October 1997.

[14] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and T. Mikkonen. The Lively Kernel: A Self-supporting System on a Web Page. In *Self-Sustaining Systems*, pages 31–50. Springer, 2008.

[15] D. Ingalls, S. Wallace, Y. Chow, F. Ludolph, and K. Doyle. Fabrik: A Visual Programming Environment. *ACM SIGPLAN Notices*, 23(11):176–190, 1988.

[16] W. R. LaLonde and J. R. Pugh. *Inside Smalltalk Volume II*. Prentice Hall USA, 1990.

[17] J. Maloney. An Introduction to Morphic: The Squeak User Interface Framework. *Squeak: Open Personal Computing and Multimedia*, 2001.

[18] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16:1–16:15, 2010.

[19] J. H. Maloney and R. B. Smith. Directness and Liveness in the Morphic User Interface Construction Environment. In *Proceedings of the 8th Symposium on User Interface and Software Technology (UIST)*, pages 21–28. ACM, 1995.

[20] J. McCarthy. *LISP 1.5 Programmer's Manual*. MIT press, 1965.

[21] Y. Ohshima, J. Maloney, and A. Ogden. The Parks PDA: A Handheld Device for Theme Park Guests in Squeak. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 370–380. ACM, 2003.

[22] S. P. Reiss, J. N. Bott, and J. J. La Viola. Plugging In and Into Code Bubbles: The Code Bubbles Architecture. *Software: Practice and Experience*, 44(3):261–276, 2014.

[23] D. A. Smith, A. Kay, A. Raab, and D. P. Reed. Croquet - A Collaboration System Architecture. In *Proceedings of the Conference on Creating, Connecting and Collaborating Through Computing (C5) 2003*, pages 2–2. IEEE, Jan. 2003.

[24] R. B. Smith. Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic. In *ACM SIGCHI Bulletin*, volume 17, pages 61–67. ACM, 1987.

[25] M. Taeumel, M. Perscheid, B. Steinert, J. Lincke, and R. Hirschfeld. Interleaving of Modification and Use in Data-driven Tool Development. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, pages 185–200. ACM, 2014.

[26] S. L. Tanimoto. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming (LIVE)*, pages 31–34. IEEE, 2013.

[27] D. Ungar and R. B. Smith. Self. In *Proceedings of the 3rd Conference on History of Programming Languages (HOPL)*, pages 9/1–9/50. ACM SIGPLAN, 2007.

APPENDIX

A. LINES OF CODE

Here is the code that was used to count the classes, methods, and source code lines:

```
| morphic mvc tools other target cls
morphicPrefix mvcPrefix toolsPrefix |
morphic := #(0 0 0 morphic).
mvc := #(0 0 0 mvc).
tools := #(0 0 0 tools).
other := #(0 0 0 other).

morphicPrefix := 'Morphic-'.
mvcPrefix := 'ST80-'. "Older versions use 'Interface–'."
toolsPrefix := 'Tools-'. "Newer versions only."

SystemOrganization categories do: [:cat |
   (cat beginsWith: morphicPrefix)
      ifTrue: [target := morphic]
      ifFalse: [(cat beginsWith: mvcPrefix)
         ifTrue: [target := mvc]
         ifFalse: [(cat beginsWith: toolsPrefix)
            ifTrue: [target := tools]
            ifFalse: [target := other]]].

   "Count number of classes."
   target at: 1 put: (target at: 1) +
   (SystemOrganization listAtCategoryNamed: cat)
   size.
   "Count number of methods for each class."
   (SystemOrganization listAtCategoryNamed: cat)
   do: [:className |
      cls := Smalltalk at: className.
      target at: 2 put: (target at: 2) + cls
      selectors size + cls class selectors size.
      "Count the lines of code for each method."
      cls selectors do: [:sel | target at: 3 put:
      (target at: 3) + 1 + ((cls sourceCodeAt:
      sel) count: [:char | char = Character cr])].
      cls class selectors do: [:sel | target at: 3
      put: (target at: 3) + 1 + ((cls class
      sourceCodeAt: sel) count: [:char | char =
      Character cr])]].
].

(morphic, mvc, tools, other) inspect.
```

It works in all Squeak versions. Note that we use class categories to distinguish between classes related to MVC, Morphic, or the rest of the system. In the first Squeak versions, these categories were "Interface-*" for MVC and "Morphic-*" for Morphic. Later, MVC code was moved to "ST80-*" indicating that the code is from Smalltalk-80.

Note that we do not consider extension methods, which can be in classes of any other package, because this is only a small fraction and earlier Squeak versions did not have them.

B. THE SHELL ON TOP OF MORPHIC

The Squeak Shell can be started within Morphic by evaluating the following expression in a workspace:

```
| world |
world := SqshController new.
[world doOneCycle] repeat.
```

This works as long as there is no exception raised in the UI process, which will then let Morphic start its own loop again. When the Squeak Shell is running in Morphic, the call stack looks like this:

C. COUNTING PRIMITIVE CALLS

The meta-object protocol in Squeak supports iterating over all classes (resp. behaviors) and methods. Each method is an instance of `CompiledMethod`, which provides access to its byte codes, literals, and method header. This information can be used to find out about primitive calls into the VM:

```
| methodsPrimitive methodsPlugin |
methodsPrimitive := OrderedCollection new.
methodsPlugin := OrderedCollection new.
SystemNavigation default
  allSelectorsAndMethodsDo: [:beh :sel :method |
    "Primitives 256 to 519 indicate quick return of fields."
    (method primitive > 0) & method isQuick not
      ifTrue: [
        method pragmas first arguments size = 1
          ifTrue: [methodsPrimitive add: method]
          ifFalse: [methodsPlugin add: method]]].
```