Interleaving of Modification and Use in Data-driven Tool Development

Marcel Taeumel[1] Michael Perscheid[2] Bastian Steinert[1] Jens Lincke[1] Robert Hirschfeld[1]

[1]Hasso Plattner Institute, University of Potsdam, Germany
[2]SAP Innovation Center Potsdam, Germany
[1]{first.last}@hpi.uni-potsdam.de [2]michael.perscheid@sap.com

Abstract

Programmers working in a Unix-like environment can easily build custom tools by configuring and combining small filter programs in shell scripts. When leaving such a text-based world and entering one that is graphics-based, however, tool building is more difficult because graphical tools are typically not prepared to be easily re-programmed by their users. We propose a data-driven perspective on graphical tools that uses concise scripts as glue between data and views but also as means to express missing data transformations and view items. Given this, we built a framework in Squeak/Smalltalk that promotes low-effort tool construction; it works well for basic programming tools, such as code editors and debuggers, but also for other domains, such as developer chats and issue browsers. We think that this perspective on graphical tools can inspire the creation of new trade-offs in modularity for both data-providing projects and interactive views.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments—graphical environments

Keywords Vivide, graphical tools, tool building, scripting, adaptation, reflection

1. Introduction

Integrated programming environments such as Eclipse and Visual Studio represent modular and extensible software systems par excellence. Programmers working in such environments can choose from a prolific variety of plugins, which support activities such as program understanding, modification, and deployment. Their main challenge is to master the graphical user interface, that is, learn about features, shortcuts, and best practices. Even this learning effort is manageable because plugin-based tools can be configured and combined to accommodate many different domains, tasks, or personal preferences. Given this, programmers can conveniently take on the role of tool users.

Building tools comes with a subjective trade-off between utility and usability. Thus, programming tools are likely to exhibit deficiencies during actual usage. When tool builders assume an idealized set of prospective tasks and users, there is a chance that they make inadequate assumptions or miss some corner cases. In that case, the tool user who detects such a deficiency can contact the tool builder; bug notices or feature requests can typically be submitted. Then, the user can wait for a resolving response or go on working around the detected deficiency. Now for *programmers* being the tool users, this procedure may be unsatisfactory. If they have access to the tool's sources, they may want to address the problem by themselves to save time.

However, building graphical tools is a challenging endeavor. Even simple tools require much code to be written because frameworks such as Eclipse/RCP[1] and Qt/UI[2] impose verbose patterns. Typically, that code describes the rules of projecting data to graphical widgets and of injecting changes back to it; but the unfamiliar reader can have a hard time ([26], [7], [20]) localizing and understanding those rules beneath numerous packages, classes, and methods. The adaptation process will be impeded if programmers are not versatile enough to be both tool user *and* tool builder.

Consequently, the sole means of configuring and combining plugins in graphical environments is too limited and many programmers may hesitate to employ their skills for adaptation purposes because diving into tool source code is distracting and time-consuming. Indeed, we made promising observations in a different kind of programming environment where graphical interactivity is of minor but programmatic ease of major interest. Consider the following example of a Unix shell script:

Onward! '14, October 20–24, 2014, Portland, OR, USA.

ACM 978-1-4503-3210-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2661136.2661150

[1] Eclipse Rich Client Platform, `http://eclipse.org`

[2] Qt User Interfaces, `http://qt-project.org`

```
  curl http://en.wikipedia.org/wiki/Unix \
| grep -o -P 'href="/wiki/.*?"' | sort | uniq \
| sed -r 's/href="(.+)"/http:\/\/en.wikipedia.org\1/g' \
> urls.txt
```

This is a script that will execute in many shells on Unix-like systems such as in *bash* on *Linux*. It retrieves the HTML content of the Wikipedia article about Unix (`curl`), extracts relative URLs to other articles (`grep`), sorts them (`sort`), removes duplicates (`uniq`), transforms relative URLs into absolute ones (`sed`), and writes the output into a file (`urls.txt`).

In such text-based environments, programmers can benefit from tools that follow the *filter pattern* [31]. These filters are not interactive but transform text streams, which origin from files or processes, according to simple configuration and combination rules. Interactive shells and their scripts represent powerful interfaces that enable programmers to employ filters in daily work. They can create and adapt respective tools with low effort while directly working with relevant data.[3]

Based on this observation, we want to address the following research question:

> How can we support programmers to build graphical tools with an efficiency comparable to Unix' filters while retaining the virtues of graphical components?

We do not just want to embed a command-line interface into a graphical environment. Many environments already have that. Rather, we are looking for a mechanism that supports configuring and combining interactive views ranging from common list-based widgets to advanced software visualizations such as [8] and [14]. In that sense, we want to support programmers to focus on their relevant software artifacts while employing rich views. In general, this should improve the support for iterative, data-driven tool development and programmers would be encouraged to directly try out any idea whether it might turn out beneficial or not. They would be in control of how to explore and modify the accessible space of software artifacts to fulfill the current programming task efficiently.

In this paper, we present a mechanism that supports low-effort construction of graphical tools. It resides between the fields of (1) processing data and (2) presenting data in graphical views on screen. On the one hand, it can provide the necessary glue to decouple both fields and promote their extensibility and reusability. On the other hand, it can reach into one or the other field to provide missing functionality ad-hoc according to specific domains, tasks, or personal preferences. Basically, it is a *script-based, data-driven* approach to *transform* software artifacts and *prepare* them before showing them on screen. We think of graphical tools as multiple pipelines where artifacts flow through and where programmers can look into to explore and modify them. Still, the level of interactivity is mainly controlled by existing views and the kind of available data mainly by the rest of the environment. Currently, we are investigating how far our mechanism can reach into both fields to provide a better trade-off between complexity and flexibility.

In this paper, we make the following contributions:

- A data-driven perspective on graphical tools that forms a simple conceptual model for programmers with the intent to provide frameworks for low-effort tool construction
- An concrete mechanism for scriptable tool construction in Squeak/Smalltalk[4] that considers our data-driven perspective
- An example case that illustrates how programmers can perform comprehension activities while iteratively building and adapting supportive tools

In the next section 2, we describe problems of traditional designs for graphical tools and shift the focus from behavior-driven to data-driven architectures. In section 3, we describe our mechanism for script-based, data-driven tool construction. We present a bigger example that shows applicability of our framework in section 4. A discussion about the tool building effort follows in section 5. Finally, section 6 gives an overview of related work and section 7 concludes our thoughts.

2. From Behavior-driven to Data-driven

We describe our notion of integrated programming environments as a basic argument for further ideas and conclusions in this paper. Then, we identify problems in traditional, behavior-driven tool designs and promote a data-driven perspective on graphical tools.

2.1 Live Programming Systems

Programmers benefit from short feedback loops. Such loops usually consist of code reading, writing, compiling, and running. One goal of programming environments is to integrate those programming activities in a way that programmers barely notice context switches but can focus on their tasks. As those tools are influenced by the underlying programming language and execution environment, this works, for example, arguably better for Java projects in Eclipse than it does for C++ projects in Visual Studio. Then, there are so-called *live programming systems*, which take this idea one step further and allow for actually molding running programs, which effectively provides a sense of *immediate feedback*. Prominent examples include Self [37], Squeak/Smalltalk, and LivelyKernel [22]. Considering the tool building scope of this paper, we continuously think of such reflective systems where the base effort is arguably lower compared to extending Eclipse or Visual Studio.

[3] The Microsoft PowerShell provides cmdlets, which brings this idea from the file-based to the object-oriented world.

[4] The Squeak Programming Environment, `http://www.squeak.org`

Squeak/Smalltalk is an object-oriented, class-based, dynamically typed programming language and environment. Typical for Smalltalk implementations, code writing and code execution go hand in hand. Programmers are able to evaluate a piece of Smalltalk code in any standard tool that provides a text input field such as code browsers or debuggers do. Having this, the process of creating a program interleaves with the process of debugging and using it. Additionally, Squeak provides a graphics framework, which takes advantage of the environment's molding capabilities: *Morphic* [24].[5] The run-time state of any graphical component, called *morph*, can be accessed with a simple mouse click if it is visible on screen as part of the running program. Basically, the modified program can then immediately exhibit its adapted behavior.

However, the flexibility of the Squeak/Smalltalk programming system did not yet yield a mechanism to promote low-effort tool construction per se. We argue that, still, many tools employ designs that stretch the feedback loop unnecessarily. Typically, there is no direct connection between a tool's run-time components and its source code [9], modifications involve domain-independent redundancies, and there have to be additional means to reliably update tools to keep on using them [21].

2.2 Simplifying Behavioral Descriptions

The Unix filter pattern [31] emerged from the idea to write simple, reusable C programs with the intent to solve larger problems in small steps. Although complexity in a C program is not limited as such, programmers are advised to only read from the standard input stream, consider some command-line parameters, and eventually write the results to the standard output stream. They could introduce side effects by reading other files or talking to other processes, but this would lower the chance of their program to be reused by other programmers.

The Unix shell, then, provides an appropriate interface to take advantage of all those small filter programs. Although shells provide complete scripting languages too, their simple pipe syntax is sufficient to connect input and output streams; the resulting *scripts* are programming tools—and filters—of their own right. Encapsulating the introductory example into a script that is configurable to query any Wikipedia page and extract their absolute urls can look like this:

```
#!/bin/bash
  curl http://en.wikipedia.org/wiki/$1 \
| grep -o -P 'href="/wiki/.*?"' | sort | uniq \
| sed -r 's/href="(.+)"/http:\/\/en.wikipedia.org\1/g'
```

Now, programmers can use this script `wiki.sh` in another script or directly at the command line like this:

```
./wiki.sh Unix > urls.txt
```

[5] Note that the idea of Morphic has a previous implementation in Self [37] and a more recent one in the JavaScript Web programming system called LivelyKernel [22].

We appreciate the simplicity of describing such tool behavior. In short, we want to support the following characteristics in graphical tools:

Simple means of configuration Many filters accept parameters from the command line such as switches or regular expressions to configure their internal behavior.

Simple means of combination The shell provides a simple syntax to connect a stream of data between filters (i.e. "|") and to redirect streams from and to files (e.g. ">").

Simple means of abstraction Writing and reusing shell scripts is straightforward. Only a small subset of the scripting language is used to, for example, access parameters (e.g. "`$1`"), read from the standard input (i.e. "`read`"), or write to the standard output (i.e. "`echo`").

Data-driven usage When using filters, programmers can focus on their actual software artifacts, which mainly represent files and their text-based contents.

In general, we think that such a mechanism is simple yet powerful enough to solve a vast range of problems by giving programmers the chance to employ their skills for unanticipated programming tasks. Programmers can take on both roles of a tool user and a tool builder. Eventually by simplifying behavioral descriptions and variations in such a way, the valuable data comes to the fore.

In Squeak/Smalltalk, there already is a comparable programming interface to work with collections of objects. Consider our Wikipedia example using Smalltalk syntax:

```
((WebClient httpGet:
  'http://en.wikipedia.org/wiki/Unix') content lines
    gather: [:line |
      'href="/wiki/[^"]*"' asRegex matchesIn: line])
    asSet asOrderedCollection "Remove duplicates."
    sorted "Lexical sort."
    collect: [:url |
      (url copyReplaceTokens: 'href="'
           with: 'http://en.wikipedia.org')
             allButLast "Remove trailing quote."].
```

Basically, it performs the same steps as the shell script; the result is a list of string objects. However, this functional composition of message sends and parameters can easily produce complicated source code when working with software artifacts. This example should indicate, that building graphical tools in Squeak/Smalltalk is not straightforward—even if there is such a simple yet powerful language concept, that is, objects sending and receiving messages. Many tool frameworks for Smalltalk, such as [1] and [4], dictate complicated patterns where data processing is overshadowed by source code for graphical widgets. *Can there be comparable data-driven semantics for graphical tools?*

2.3 A Data-driven Perspective on Graphical Tools

We see graphical tools as data processing pipelines whose intermediate results can be displayed on screen. Software artifacts are repeatedly transformed and prepared for views;

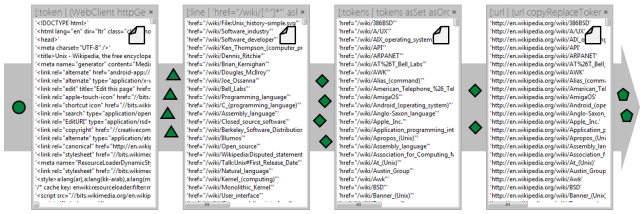


Figure 1. Our data-driven perspective where programming tools are pipelines processing scripts to transform software artifacts and prepare them for views.

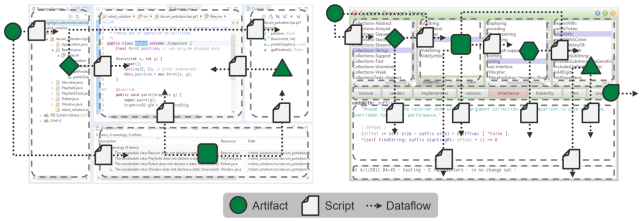


Figure 2. Our data-driven perspective applied to Eclipse (left) and Squeak (right). Software artifacts include projects, files, classes, and methods.

programmers interactively explore and modify artifacts through such views. Means of *configuration* represent the selection of relevant artifact relationships and the extraction of characteristic information to reveal an appropriate degree of insight. Means of *combination* represent the arrangement of multiple views, each having particular strengths, that cooperate and help programmers see problems from different angles. Means of *abstraction* represent different groups of tools and the notion of tool boundaries like it exists in terms of Eclipse's perspectives.

The basic idea is illustrated in Figure 1 using our introductory example; intermediate results are visualized using list-based widgets. Figure 2 applies our data-driven perspective to the programming environments Eclipse and Squeak; each environment consists of rectangular boxes that exchange software artifacts and where scripts prepare them for visualization on screen.

By projecting this data-driven perspective on graphical tools, we support programmers to focus on their domain-specific software artifacts. In contrast to usually not self-explaining interfaces, data-driven tools provide discoverable cues for the whereabouts and happenings of software artifacts. For example, a typical user may reason about the rules of practice in a graphical user interface like this:

> If I click on *that file name* on the left-hand side, the environment *somehow* shows that file's contents in the central, editable area. I have to *remember* that.

In our data-driven perspective, we anticipate thoughts that focus on artifacts and their projections like this:

> If I choose *that file representing my module* in the left-hand view, this very artifact will flow to the central, editable view where it is *projected* to its text-based contents. I can *change* that if I want to.

There are already list-based views where programmers can be in control of their software artifacts because each artifact has a distinguishable representation on screen. In our perspective, we can also make the rules of processing such artifacts *explicit and customizable* because we establish *discoverable boundaries* in the graphical user interface.

3. The Mechanism

In this section, we apply our data-driven perspective to describe our mechanism that promotes low-effort construction of graphical tools. Our project is available on GitHub: `https://github.com/marceltaeumel/vivide`.

We implemented our framework in Squeak and the following examples will therefore consist of Smalltalk code. As our inspiration comes from Unix filters and shell programming, we comparably describe our means of configuration, combination, and abstraction.

3.1 Configuration: Exploring Data and Views

One difficult challenge for tool builders is to select appropriate data and appropriate views according to anticipated domains, tasks, and users. That is why we choose *scripts* as means of configuration [29] to decouple software artifacts, which represent our data, from graphical widgets, which represent our views. We assume both artifacts and widgets to be, in some extent, already available in this setting. Basically, tool configuration via scripts is two-fold: programmers can (1) extract interesting *properties* from artifacts and they can (2) try out different ways to map these properties to *features* of widgets. Consider the following script that extracts the properties `selector` and `timestamp` from methods and maps them to the features `text` and `tooltip`:

```
[:method | { #text -> method selector.
             #tooltip -> method timestamp }].
```

When our framework evaluates such scripts on concrete artifacts,[6] it generates *models* with nodes for artifacts having interfaces as defined in the scripts. For this example, widgets can retrieve a text-like property[7] and a tooltip-like property from the particular model—if they need other data, they will have to fall back to defaults. But if programmers know about those widget's expectations, they can accommodate by modifying script code right away.

Script evaluation will be triggered *immediately* if scripts change or if artifacts change thus providing a short feedback loop. To detect artifact changes, scripts can be registered to

[6] Scripts are Smalltalk blocks that can receive artifacts either one-by-one or all-at-once.

[7] We will not distinguish between properties and features but use the term *properties* for the remainder of this paper.

event sources. For example, if programmers write a script about methods, as listed above, they can use Squeak's *system change notifier* to tell our framework about changes, which will then re-evaluate that script, update the model, and hence allow widgets to update their contents, too.

Programmers are free to choose from or switch to any existing script and widget when facing concrete data. While scripts can differ in their data processing rules, widgets can differ in their graphical richness and interactivity as illustrated in Figure 3. As for the mechanics, different widgets will just to talk differently to our generated model; if needed, programmers can still adapt the script to effectively adapt the model's interface.

We think that scripts should not only encapsulate data projection into widgets but also injection of changes from widgets back to data sources. That is why the generated model can not only provide properties but also accept values that change properties—just like "setters" mirror "getters" in abstract data types. Consider a modified version of the example above that now has a rule to rename methods:

```
[:method | #text -> [method selector]
                <- [:value | RenameMethodRefactoring
                                on: method to: value] ].
```

One benefit of using Smalltalk as a scripting language is that programmers can easily store *code blocks* in the model. Code allows widgets to update their contents without having to re-evaluate scripts and, more importantly, to modify artifacts. As Squeak provides *closures* to bind abstract identifiers to concrete objects, programmers can express the rules of reading and writing software artifacts in one place.

In the example above, the artifact `method` is closured in the respective blocks. Whenever widgets *read* the text-property, the first block is evaluated. Whenever widgets *write* into the text-property, the second block is evaluated with that new value. Any side effect introduced by the script, such as by calling a refactoring engine, is not analyzed by our framework.[8]

If scripts have to access multiple artifacts that are not directly related, our framework includes means to provide them. For example, a piece of text may represent method source code whose modifications have to be compiled back into its class; a script can look like this:

```
[:class :code |
     #text -> code
           <- [:newCode | class compile: newCode] ].
```

Many tools support browsing some data graph, that is, they exploit relationships between particular software artifacts. This means that scripts should not only prepare data for views but also perform intermediate *transformations*. For example, showing all methods of a class requires transforming that class to its methods first; a script can look like this:

[8] However, precautionary script re-evaluation after writing into the model can ensure updated contents in the tool.

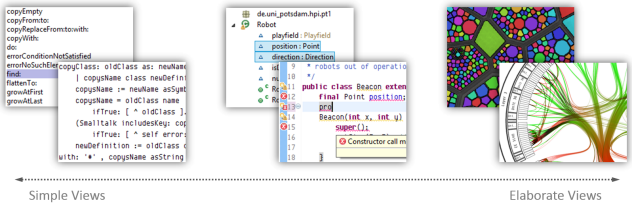


Figure 3. Views on software artifacts vary in graphical richness and level of interactivity. Tools range from providing plain text only (left) over additional layers of information (middle, Eclipse) to very elaborate visualizations (right, [14] and [8]).

```
[:class | class methods collect: [:method |
  { #object -> method.
    #text -> method selector.
    #tooltip -> method timestamp }] ].
```

In this case, we need to include the `object` property because our model stores references to the actual software artifacts for *combining scripts* across multiple widgets. Nevertheless, there is syntactic sugar for that, which we will use in the remaining examples:

```
[:class | class methods].
[:method | { #text -> method selector.
             #tooltip -> method timestamp }].
```

Effectively, that script does the same things but it hides some redundancies by separating *object transformation* from *property extraction* steps. Our framework uses the upper block to transform classes to methods; then it flattens all results[9] into one list and evaluates the second block for each method to extract its properties into the intermediate model. The `object` property can now be inferred.

The data that is projected via model properties into widgets can differ in their complexity. With simple properties we mean strings or numbers or any piece of information where displaying is straightforward. But there may be widgets that can handle more complex artifacts without having to prepare them in scripts. A method editor widget, for example, may accept a method (complex) without having to extract its source code (simple) beforehand. However, we are not in favor of complex, monolithic widgets but rather small, combinable ones. Given this, we think of complex properties being *additional widgets* embedded in other ones. For example, imagine a widget that arranges its items in a two-dimensional grid. A script can provide those items:

```
[:employee | #item -> (CircleMorph new
                        radius: employee salary / 100;
                        x: employee numberOfProjects;
                        y: employee age;
                        tooltip: employee name) ].
```

The result would be a two-dimensional point cloud that may reveal correlation of various employee traits. Again, the Squeak environment makes it easy to create graphical

[9] Smalltalk blocks implicitly return the result of their last expression.

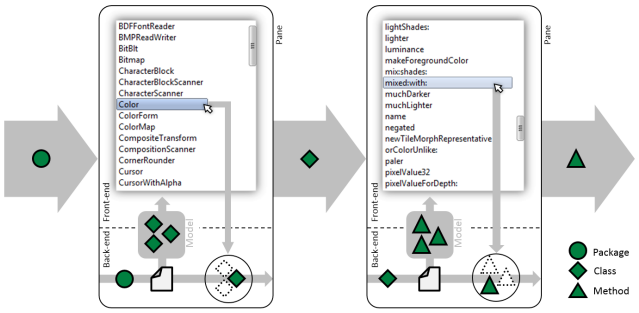


Figure 4. In our framework, graphical tools consist of cooperating panes, which encapsulate interactive widgets and evaluate scripts on incoming artifacts.

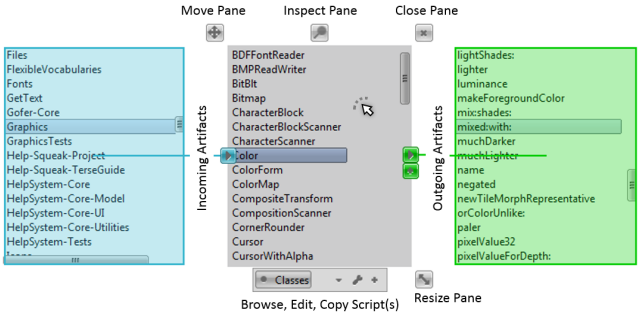


Figure 5. With a single mouse click, a halo over that pane appears and reveals buttons to access and modify scripts, dataflow, and layout.

objects (here: `CircleMorph`) in scripts. We see this way of embedding widgets in widgets as a first idea for providing *means of abstraction* in graphical tools.

Given some software artifacts for the data input and some widgets for the graphical output, we have that notion of a *script* and a *scripting language* to describe projection and injection rules in between. The *generated model* represents the data structure that has the interface as expected from a particular widget. This forms the basis for configuring graphical tools in our mechanism.

3.2 Combination: Establishing Dataflow

Tool builders are responsible for decomposing the graphical interface into multiple, interacting widgets. Typically, there are buttons, lists, charts, or text fields in such tools that cooperate to help users accomplish particular tasks efficiently. Our mechanism supports this process of decomposition by means of combining multiple scripts and their widgets to establish dataflow.

We provide *panes* as uniform building blocks, which describe where to show information on screen. That is, panes are invisible rectangles with a position and an extent being placeholders for actual content. Encapsulated in each pane, there is set of artifacts, a current script, and a current widget. When new artifacts arrive, the particular pane evaluates its script and updates the intermediate model for its widget.

Panes can talk to each other as illustrated in Figure 4. Such communication allows for modeling the dataflow within one tool and across tools. For each pane, there is an *interaction loop* through the widget, which allows users to influence the dataflow. They can, for example, select which artifacts to process if widgets support such a selection like most list-based ones do.

Like user-defined selections, there can be other situations where widgets may want to talk to our *framework*. This is different from the intermediate model, which represents a scriptable interface to the *artifacts*. The interface between widgets and our framework is not scriptable and we argue that this is not necessary because widgets are unit of reuse. Preparing a widget to be used in our framework is considered a one-time effort. As mentioned above, the `object` property in the model stores a reference to the artifact; widgets do only know about model nodes and not about the artifacts.

If we want tool users to be also tool builders, there should be a simple way to switch between both roles. That is why we provide direct access to tool components, that is, panes with scripts and dataflow settings. In Squeak, we can employ *halos*, which represent a user interface to explore and edit properties of graphical items, so-called *morphs*. A dedicated user input, typically just a mouse click, reveals a pane's custom halo as shown in Figure 5:

Layout Programmers can modify a pane's position and extent to effectively control the space that widgets occupy.

Scripts Programmers can access and modify the current script as well as the pane's script database.

Dataflow Incoming connections are shown in blue at the left side of a pane. Outgoing connections are shown in green at the right side of a pane. Programmers can add or remove such connections via drag-and-drop or click.

Internals Programmers can inspect a pane's internals for debugging purposes. This represents a link to the underlying Squeak environment.

This part of our mechanism does not depend on the reflective capabilities of live programming environments. Many window managers already provide elaborate means of configuring and combining the layout of graphical tools.

The more complex a tool is, the more complex is also its dataflow. It may not be just one linear stream of data anymore. In Unix, there is a filter called *tee*, which allows for forking streams. Consider the following example that downloads a file and also stores its SHA-1 hash value while doing so:

```
curl https://www.debian.org/amd-64/debian-7.6.0.iso \
| tee >(sha1sum > debian-7.6.0.sha1) \
> debian-7.6.0.iso
```

Our framework supports forking and joining dataflow to some degree. A pane can have multiple incoming and outgoing connections from and to other panes. Scripts will re-

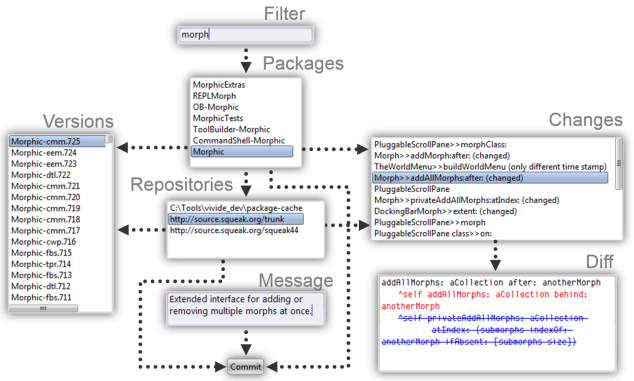


Figure 6. Exploded view of a code commit tool. Arrows illustrate forks and joins of dataflow between widgets.

ceive *tuples of artifacts* in the order of the connections. If panes provide more than one artifact, n-tuples will be formed like the *Cartesian product* in set theory. As our scripts are Smalltalk blocks, appropriate blocks have to support sufficient parameters. Imagine one pane providing classes and another one providing categories; a script can transform both into methods like this:

```
[:class :category | {
  class.
  class organization listAtCategoryNamed: category }].
[:class :selector | class compiledMethodAt: selector].
```

The upper part takes a tuple of $(class, category)$ and implicitely transforms it into many tuples of the form $\{(class, sel_1), \ldots, (class, sel_n)\}$. The lower part takes all tuples and transforms them into actual method objects.

For a bigger example, consider Figure 6: A tool for storing source code in a remote repository has more complex dataflow characteristics. The user has to make three inputs before hitting the *commit*-button: choose a package, choose a repository, and enter a message. All three artifacts are necessary before the commit action can take place; the pane with that button performs the *join* of data. *Forks* are used to, for example, show versions in the repository and a diff to the head version. The scripts are listed in the appendix.

Panes can manage multiple scripts. When new artifacts arrive, there is a dispatch between scripts according to some associated type information. This means that tools can be designed to reuse certain panes by writing scripts for particular kinds of artifacts. For example, the standard Squeak class browser (Figure 2) has four different ways of showing code in its big text box: a template for a class definition, an actual class definition, a template for a method definition, or an actual method definition depending on whether a class category, a class, a method category, or a method flows into that pane.

Given the panes as uniform and interacting building blocks configurable via Morphic halos, programmers can combine them to build tools with flexible dataflow patterns.

3.3 Abstraction: Graphical Tools as "Gray Boxes"

Typically, abstraction means hiding details and reducing one's knowledge to some rough understanding. *Names* are sufficient to think about further ways of configuring and combining abstract concepts. Different layers of abstraction support modularizing complex tools; one does not always have to remember every detail.

Unix filters represent such named abstractions. They can be C programs or other shell scripts; programmers do not bother but focus on the general purpose, input, and output. We call this a *black-box* perspective.

We argue that, for the graphics-based world, information hiding is neither fully possible nor desired. There is always something to *see* from graphical tools that are used by other—more abstract—graphical tools. For example, *wizard dialogs* are typically used to guide users through some complicated configuration process, but another tool will then build upon those inputs. Graphical tools are appreciated for their visuals and interactivity; hiding those virtues seems unreasonable. At most, one could take on a *gray-box* perspective where some internals are hidden but the graphics are not. Still, abstraction is important because script code and dataflow characteristics can get quite complex and tools can benefit from proper modularization.

Our mechanism has two complementary means of abstraction: one for scripts and one for panes. In conjunction, they allow for hiding details and reusing tool components created with our framework.

Each script can have an *identifier* to reference it in other scripts. Such references are resolved in a script database, which can be local to a pane, a tool, or the whole environment. For example, the following script extracts some default properties (text, tooltip, icon) for artifacts:

```
[:object | {
  #text -> object asString.
  #tooltip -> object class asString.
  #icon -> (object isMorph
    ifTrue: [object imageForm scaledToSize: 16@16]
    ifFalse: ["No icon."]) }].
```

Now, we give that script the identifier *defaultProperties* and use it in another script that extracts all children[10] of a morph:

```
[:morph | morph submorphs].
#defaultProperties.
```

This way of abstracting from script code is similar to messages in object-oriented code, which abstract from actual method implementations. For now, we use a single database to look up script identifiers so we can share them between all tools in the environment. This eases emergent tool design as described later in this section.

Besides hiding script code with script identifiers, our framework supports grouping and encapsulating multiple

[10] We discuss the creation of hierarchical models in section 5.

panes as graphical means of abstraction. We provide so-called *multi-pane widgets* to manage such groups of panes. Such widgets expect a model that provides scripts, layout information, and dataflow specifications. For example, a script that produces such a model for configuring the three panes as in Figure 5 can look like this:

```
[:object | {
  #id -> #left. #in -> #(outer). #out -> #(middle).
  #script -> #packages. #bounds -> #(0 0 200 250) }].
[:object | {
  #id -> #middle. #in -> #(left). #out -> #(right).
  #script -> #classes. #bounds -> #(200 0 200 250) }].
[:object | {
  #id -> #right. #in -> #(middle). #out -> #(outer).
  #script -> #methods. #bounds -> #(400 0 200 250) }].
```

These are three subsequent property extraction blocks in one script to describe three panes. In the model nodes, those properties will not overwrite each other but get a running number such as `id_1`, `id_2` and `id_3`.

The properties `id`, `in`, and `out` specify the dataflow. As mentioned above, panes can have multiple incoming and multiple outgoing connections to other panes. The pane identifier `#outer` is reserved and refers to the outer pane, which contains this multi-pane widget. The property `script` refers to a script identifier that will be looked up, which can be a specification for another multi-pane widget. Note that every script is associated with a *preferred widget* to be initialized with in panes; it is a multi-pane widget in this case. Programmers can change that. They could also use other widgets for this script but those may not find any useful data in the generated model.

In our framework, the outermost pane is embedded in a window, which represents a tool boundary. There can be many windows that have such panes as their sole contents. This is specific for Squeak and not part of our mechanism; other environments may provide different high-level tool containers. Indeed, panes can talk to each other across container boundaries. Such data-driven communication channels can be established ad-hoc (Figure 5). We argue that tool integration becomes straightforward.

Our mechanism describes a kind of *graphical abstraction* layer that can be inserted between other any other one. At the one end, a pane being a graphical item is part of some container. At the other end, a pane itself is a container for other graphical items. There can be an arbitrary alternation between different implementation strategies in the whole graphical hierarchy. Widgets do not have to be aware of being host for some panes as long as they provide means to embedd new graphical items. For example, the following script provides additional panes for each artifact:

```
[:object | #item -> (Pane new
                      objects: {object};
                      script: #defaultProperties) ].
```

Our mechanism also describes a kind of *programmatical abstraction* layer. The scripts are evaluated with data-driven semantics; within each script, there is object-oriented

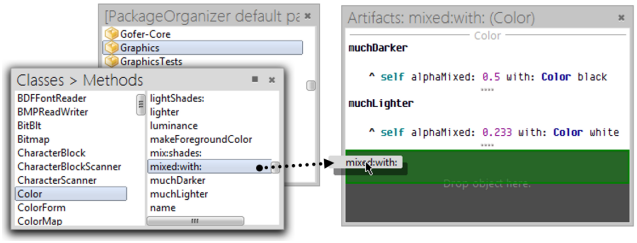


Figure 7. Panes embedded in overlapping windows. Regular widgets (left) support dragging and our artifact list widget (right) supports dropping and hence cherry picking artifacts.

code. Even without creating graphical tools, programmers can switch between both paradigms as needed. Consider the following example that evaluates a script from within object-oriented code:

```
| script colors morphs |
script := Script newFrom: {
  [:colorName | Color fromString: colorName].
  [:color | {color. RectangleMorph new}].
  [:color :morph | morph color: color] }.
colors := #(red green blue yellow).
morphs := script evaluate: colors.
```

Here, the script transforms color names to real color artifacts, creates pairs of color and new rectangle, and finally sets the color of these rectangles. We are still experimenting with the cooperation of both programming paradigms.

Given the flexibility of our scripts and multi-pane widgets, programmers can encapsulate tool components and reuse them in different scenarios while preserving their cohesion.

3.4 Data-driven Tool Development

We envision emergent tool designs. Our framework supports programmers to be both tool user and tool builder. Given that we keep the overhead for switching between roles minimal, there may be no dominant role anymore. When working with concrete artifacts, programmers can create and modify scripts, try out various widgets, and combine them to efficiently use the screen real estate. There is arguably the chance that tools just *happen to be built* while programmers focus on processing their domain- and task-specific data.

The traditional approach of tool building is more anticipatory. Tool builders have assumptions about target domains, tasks, and users. However, their tools are typically not meant to be extensively re-programmed by the users but only slightly adjusted. Users depend on their tool builders for making bigger adaptations; our mechanism tries to remove exactly that dependency and improves unanticipated scenarios.

We propose a tool building approach that allows for *dynamically* working with artifacts, widgets, panes, and scripts. Tools are molded iteratively while they are running. At the beginning, programmers may only have a single artifact at hand and evaluate an appropriate script, be it freshly

written or looked up in a database, on that artifact. Through continuous interaction with widgets and scripts, a concrete design for that graphical tool can emerge.

Cherry picking interesting artifacts can further promote such unanticipated tool building activities as shown in Figure 7. We created an *artifact list widget* that supports a way of collecting and arranging artifacts via drag-and-drop. Effectively, this widget injects new artifacts into the pipeline. Scripts can associate artifacts to graphical items—such as interactive editors—that will be arranged in the widget:

```
[:object | #item -> (object isCompiledMethod
                        ifTrue: [MethodEditor]
                        ifFalse: [TextEditor]) ].
```

In our implementation, dropping artifacts behind all windows on the background spawns a new window that has a single pane with such an artifact list widget. Using the pane's halo (Figure 5), programmers can start writing scripts for the collected artifacts and try-out different widgets. The process of unanticipated tool building may begin [35].

4. Example Case: Program Comprehension

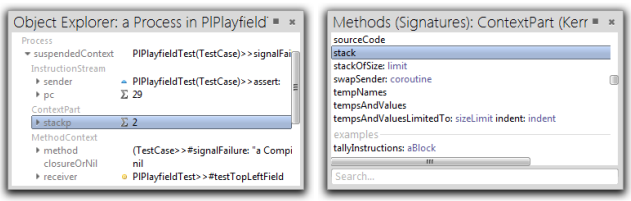
In this section, we explain a case about program comprehension to emphasize the advantage of having a data-driven perspective on graphical tools and our framework that supports low-effort tool creation and adaptation. The challenge of comprehending abstract source code in complex software systems is part of many programming tasks ([19], [32]).

4.1 Setting

The programmer works in the Squeak/Smalltalk environment. The domain in this example is a game where the player has to quickly place pipelines on a playfield's grid to carry a stream of water from a spring to a drain without spilling it all over the field.

The story has three parts: (1) traditional debugging, (2) run-time tracing, and (3) social collaboration. Typically, the execution of a program is interrupted if something goes wrong; a single execution state can be explored. Then, there are research projects that support browsing multiple execution states such as [30]; those have to be integrated. Finally, a colleague or the author of a system part can be asked [34] to balance problem solving efforts.

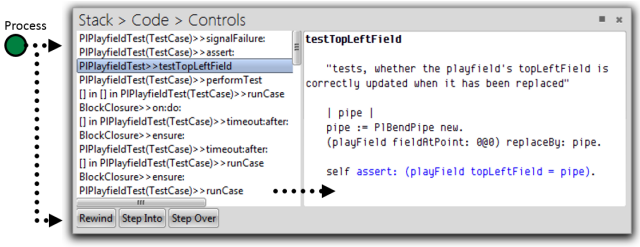
Consider the following restriction to better understand the programmer's decisions. At the beginning, there are means of executing Smalltalk code in a workspace and only two tools available: one for exploring object state (left) and one for browsing objects' known messages (right):



4.2 The Conventional Programmer

A test execution fails. The corresponding *process* is suspended and the programmer can explore its state. As there are many distracting details, she decides to write a script that extracts the current *call stack*. In that list, she discovers the name of the failing test: `testTopLeftField` in `PlPlayfieldTest`. Now, she wants to browse the *source code* for each call in the stack and therefore writes another script for a text box. There, she discovers that the final assertion fails; the prior replacement of pipes must have been failed. She wonders whether `replaceBy:` works as expected and decides to look into that call. For this, she has to control the process object and hence writes a script that allows for *unwind*, *step into*, and *step over* stack frames. Within that method, a suspicious check `canBeReplaced` on the incoming pipe object seems to work fine.

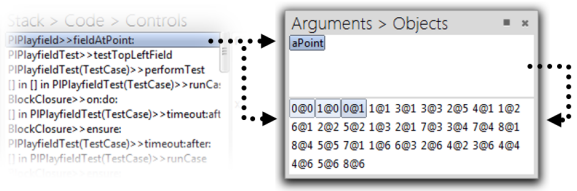
Looking at the screen, she happened to build a conventional debugger:



4.3 The Eager Programmer

She notices the argument `0@0` and wonders whether the grid starts with the point `1@1` in its upper-left corner or not. As this project seems to provide a pretty good test coverage, she tries out a recent research project [30] that allows for tracing test executions and collecting data such as receivers, arguments, and return values. Luckily, she does not have to struggle with integrating a new graphical tool but only has to write a new script for that already visible `fieldAtPoint:` call in the debugger. She extracts the *argument names* into a list to have a more convenient browser for her run-time artifacts. The research project already provides a quite convenient interface for the script. After seeing all those *point objects*, she discovers that there are other tests using `0` with success. The bug has to be somewhere else.

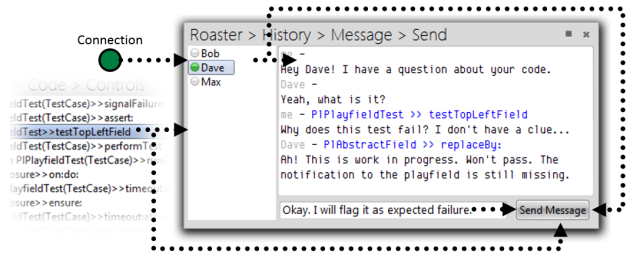
Looking at the screen, she happened to build another tool that can be connected to her previous debugger to explore arguments:



4.4 The Social Programmer

Finally, she wants to ask her colleagues who may have worked on the same part of the project. After quickly establishing an XMPP *connection* to her company's Jabber server, she writes a script that filters the *user list* by all previous authors of the selected method. As her colleagues use different nicknames for chatting than for committing code, she maps them in the script as she recalls. Only Dave seems to be available. She writes an additional script to show the chat *history*; it is stored locally. The script is registered for *notification* by the XMPP connection; on an incoming message, the history is updated and the script re-evaluated. After adding a text input field and a button for sending the *message*, she wants to start chatting. But she hesitates and remembers a research project about code chats [34] where messages can be directly associated to *code artifacts*. Having the debugger still visible, she connects its stack list to the button. In the script, she associates the message to the selected code artifact. She talks a little bit with Dave and finds out that this particular test is expected to fail. As a result of her programming task, she flags that test accordingly.

Looking at the screen, she happened to build a small chat tool that supports talking to code authors about methods:



5. Discussion

In this section, we discuss how our mechanism compares with traditional tool architectures, whether our scripting approach has reached its limits, and to which extent not-so-reflective environments may have more difficulties in providing a similar tool construction framework.

5.1 Revising the Past's Efforts

The existing tools for Squeak have a long history: much software engineering effort has been expended, many idioms and patterns have been applied. The designs range from rather monolithic ones to composite ones. Monolithic tools, on the one end, may coalesce all features in very few classes, which decreases cohesion per class and impedes code comprehension. For example, Squeak 4.4 employs only six classes to implement code browser, debugger, and object inspector with about 4500 lines of Smalltalk code.[11] Composite designs, on the other end, may specify custom classes

[11] The reader can complement an understanding of Smalltalk's expressiveness by studying comparative discussions such as `http://c2.com/cgi/wiki?JavaVsSmalltalk`

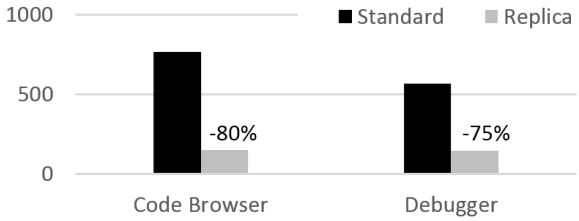


Figure 8. Lines of Smalltalk code that projects artifacts' data into widgets. We compare standard tools in a Squeak 4.4 image with our replicas.

even for single artifacts, which increases coupling per class and impedes code comprehension, too. For example, the OmniBrowser [1] framework wraps concrete artifacts in abstract nodes and the Glamour [4] framework wraps concrete widgets in abstract presentations. *How do we compare with those designs considering lines of Smalltalk code?*

We reimplemented two of the most important tools that programmers use frequently: the code browser and the debugger. Code browsers provide access to classes and methods and thus source code. Debuggers provide access to a particular execution stack and thus source code and run-time state. We only considered the projection of those artifacts to widgets and no injection of any change.

The code browser has 3 classes: `StringHolder`, `CodeHolder`, `Browser`. The debugger has 5 classes: `StringHolder`, `CodeHolder`, `Debugger`, `Inspector`, `ContextVariablesInspector`. Having this, some classes are shared via inheritance; the debugger is associated with two inspectors. For initialization, both tools use Squeak's *tool builder*, which provides a declarative interface for describing the graphical layout and callbacks.

In order to rebuild those tools, we adapted the standard tool builder to support our panes, scripts, and dataflow setup. We also created two wrapping classes to make the standard list and text widgets talk to our generated model and to our framework. In sum, we wrote about 170 additional lines of Smalltalk code in five additional classes to be prepared for building *any* graphical tool that uses standard lists or text boxes. For extracting artifacts' properties, we copied as much code as possible and also inlined methods into scripts.

In our replicas, we counted all the script code. In Squeak's tools, we looked for methods that mainly deal with artifact reading and counted all their lines. As Smalltalk methods are usually small, the risk of counting other tangled features was small, too. The actual counting algorithm performed the following steps with blocks and methods: (1) apply consistent Smalltalk code formatting, (2) remove comments and blank lines but keep method headers if any, and (3) count the remaining lines.

With our framework, we managed to reduce the amount of code up to 80%. Figure 8 visualizes the results. The main reason is their declarative, data-driven structure without having to handle intermediate representations. Then, using

many methods influences this number because each method has one line signature. For both standard tools, this results in 159 additional lines. We saved several lines because we had to merge several methods from the standard tools' code into single scripts.

The standard code browser includes many lines that prepare intermediate data structures for the widgets. This includes lists of strings and means to map indices in those lists back to their software artifacts.

The standard debugger is somewhat smaller than the code browser. Both share some classes by inheritance, but the debugger also employs composition with two inspectors, which themselves reuse code to access object state. Given this, we reused script code for inspecting object state, too.

The goal of this comparison is not primarily to show that our approach supports to reduce the code base for existing tools per se. It should serve merely as an indicator for how low the effort of building new tools can be. We do not expect programmers to replace well-designed and practical tools already available but to build custom ones tailored to particular domains, tasks, or personal preferences.

5.2 To Script, or Not To Script

Graphical tools are, basically, adapters that describe mappings between the artifacts' interfaces and the widgets' ones. We presented a framework, which allows for scripting those interfaces to support low-effort modifications. With that framework, we also introduced additional interfaces; not all of them have the property of being *scriptable*:

Artifacts ↔ Widgets The interface between artifacts and widgets is manifested in the model, which is generated by our framework using user-defined scripts. Both kinds of rules for (1) projecting data to widgets and for (2) injecting their changes back are supported.

Artifacts ↔ Our Framework As for our scripting language being Smalltalk, scripts will have full access to artifacts if those are represented as regular Smalltalk objects in the environment. If there is also an event source in the Smalltalk environment that signals object modifications, scripts will be able to register and to trigger their re-evaluation consistently.

Widgets ↔ Our Framework The interface between widgets and our framework is not scriptable but *hard-wired*.[12] In contrast to artifacts, we consider widgets as unit of reuse; interface adaptation is only required once.

For example, we created wrapping classes to reuse *pluggable widgets* from Squeak for our quantitative comparison as described above. We had to describe the rules of using our generated model and of providing user selections for our interaction loop (Figure 4). We argue that this is, to some extent, comparable to the fact that the text-based nature of terminal output in Unix environments is typically not an issue. There is no need to adjust this level of interactivity ad-hoc. Scripting is not required for this. Another example for hard-wired interactions between a widget and our framework is artifact list widgets (Figure 7): they allow for the insertion of new artifacts into the pipeline.

Scripting itself is not always straightforward in our framework. Although Smalltalk is powerful and expressive, we are still investigating ways to describe tree-structured models and artifact-independent properties. At the moment, trees with a specific depth can be described by alternating transformations with property extractions:

```
[:addressBook | addressBook persons].
[:person | #text -> person name]. "Level 1"
[:person | person friends].
[:friend | #text -> friend name]. "Level 2"
```

Recursive structures with varying, maybe infinite, depth can be described using a reference to the very same script, which effectively means a cycle during its evaluation:

```
[:morph | morph submorphs]. "id: #submorphs"
[:morph | #text -> morph name]. "Level 1..n"
#submorphs.
```

As for artifact-independent properties, there can be metadata associated with scripts and widgets may access the underlying script in the generated model. However, *constant values* can be stored in the model like the scripts for our multi-pane widgets do.

Considering the way models provide information for widgets, we also think of other ways than just mirroring. For one idea, there may be *stateful widgets* that do not discard their internal caches after each model update but rather see models as incremental information providers with a stable interface. For another, maybe complementary, idea, script evaluation may be *asynchronous*, which would better reflect the execution model of Unix' pipes and filters and which would allow for data generators and *stateful scripts*.

5.3 The Benefits of a Live Programming System

As mentioned in the beginning of this paper, we use live programming systems such as Squeak/Smalltalk as a baseline for open challenges and as primary target for our mechanism. However, one could also implement a similar framework in static, not-so-reflective, compiler-based environments. To outline the things that may be more difficult to provide in those environments, consider the following situations where having a live programming system was beneficial to us:

- There is only one space for all objects in the environment; run-time is omnipresent. Scripts can fully access and process that object space.
- Basic programming tools such as code browser and debugger can be described with scripts for `ClassDescription`, `CompiledMethod`, `Process`, or `MethodContext` instances.

[12] They are still modularized and decoupled in an object-oriented sense but programmers will have to dig into the respective code base to modify.

- We can use scripts in a meta-circular way: the rules of reading and writing scripts can be described with scripts. We treat scripts as software artifacts of their own right.
- Faulty scripts can interrupt tool updates only to a limited, comprehensible extent. Panes will display a small exclamation mark indicating the problem; programmers can debug that later with the help of continuations.
- The Morphic framework with its halo concept represents a simple user interface for modifying a tool's internals.

We argue that our mechanism can support the construction of graphical tools with low effort in other environments, even for the limited artifact spaces, more restricted scripting languages, or less interactive graphics.

6. Related Work

Our approach supports data-driven, scriptable tool creation with low effort. One important use includes programming tools that organize coherent software artifacts according specific domains, tasks, or personal preferences. Finally, our data-driven perspective touches the idea of direct manipulation interfaces.

6.1 Data-driven Application Development

The idea of data-driven approaches for building graphical applications manifested itself long time ago in the domain of *visual programming* such as Fabrik [17] and its web-based successor LivelyFabrik [23] do. The programmer can combine scriptable, graphical components and establish dataflow in between. More recent research projects include KScript [27], which employs *functional reactive programming* with declarative, data-driven constructs for building graphical applications.

There are also industry-focused projects such as [33], which combines ActiveX and JavaBeans components as filters into graphical interfaces. As spreadsheet programming is also considered straightforward, the ActiveSheets project [38] explores the possibilities of stream processing and visual output in Excel.

Our approach targets programmers but not necessarily the professional ones. We explicitly appreciate the combination of different language concepts such as object-oriented programming and data-driven scripting. In contrast to the projects mentioned, we consider means of abstraction to facilitate the construction of more complex tools.

6.2 Organization of Coherent Software Artifacts

Typically, programming tools provide views that align with the language representation. Consequently, no appropriate views exist if concerns crosscut the dominant system decomposition such as methods do that are spread across the class hierarchy. There are research projects that use semi-automatic approaches to provide those missing views.

Feature localization [13] identifies coherent source code locations that implement specific functionality in a software system. For example, UseCasePy [16] discovers from annotated acceptance tests the relevant methods that implement a particular use case. Another way of accessing relevant artifacts is to manually enhance the program description; TagSEA [36] allows programmers to attach way points like comments to the source code and provides tool support to query these for navigation. Mylyn [18] automatically captures programmers' activities and captures their working sets according to a degree-of-interest model. This knowledge reduces the information overload for similar tasks and helps developers to focus on selected software artifacts.

All those projects target also user interface integration although their main focus lies on the collection of supportive data. With our approach, we can provide means to decouple such data providers from visualizations. Programmers are free to decide where to make use of those additions in daily work.

6.3 Source Code Query Languages

There are domain-specific languages that support programmers to query the source code base for particular software artifacts. Scripts are typically exposed in the environment's user interface and can hence be used for in-situ tool adaptation.

Jquery [11] builds a fact database of the system under observation, which can later be queried in a declarative manner. The results are presented in a custom project outline view. CodeQuest [15] combines relational databases and logic programming in form of safe Datalog to provide an efficient and scalable code querying system. SOUL [10] is a logic query language for Java programs that is integrated into the Eclipse development environment. The evaluated queries are then displayed as intentional views [25] that show all artifacts of one crosscutting concern on the screen.

Those languages are not useful for configuring visual output but only for filtering the relevant data. We use Smalltalk as a scripting language and are able to describe the interface of the generated output, which will be used by widgets. Additionally, our scripts provide means to encode the rules of injecting changes back to the data; we argue that this is essential for building useful programming tools.

6.4 Direct Manipulation Programming Environments

Programming environments are extensible software systems par excellence. However, there are many research projects that question fundamental interaction and extension concepts to reduce context switches and promote direct interactions with relevant software artifacts.

Hopscotch [5] is a user interface composition framework for the Newspeak programming environment [2]. It provides an abstraction for browsing and modifying many software artifacts of interest concurrently within the same programming activity. This is difficult to achieve in tradi-

tional programming environments that either focus one artifact at a time or show many redundant information when having many windows open at once. Code Bubbles [3] is a novel user interface that summarizes working sets in form of multiple lightweight editable fragments, called bubbles, that group the different software artifacts on one screen. So, developers see all related program entities at once in order to completely focus on their specific development task. Debugger Canvas [12] extends the Code Bubbles approach to support industrial debugging scenarios. Developers debug their code within multiple bubbles representing code snippets, call paths, and parameters on an open two-dimensional pan-and-zoom interface. Also, Gaucho [28] builds on Code Bubbles and offers a complete object-oriented programming environment.

Those projects typically favor direct manipulation of artifacts but do not consider programmatic access to them. Whenever such an interaction concept interferes with the programmer's intents, adaptation is not easily possible. There are recent projects such as The Moldable Debugger [6], which supports re-programming debuggers in use. Our mechanism tackles that challenge by attaching scripts to any set of manually or automatically collected artifacts.

7. Conclusion

So far, we use our own framework and several scripts to iteratively improve the very same implementation. Being frequent users of our own approach, we formulated the following three hypotheses as a starting point for future investigations:

H1 - Applicability If programmers can modify running tools easily, none of the roles—neither tool builder nor tool user—will dominate the tooling process because each can switch to the other as needed.

H2 - Efficiency If programmers constantly modify their own tools, they will achieve a high degree of proficiency and hence need less time by making fewer errors.

H3 - Modularity If providers for data or widgets target our approach, their components will be more reusable and extensible relying on the programmers being the ones who can integrate those with reasonable effort.

Live programming systems such as Squeak/Smalltalk provide short feedback loops to promote iterative, low-effort, and high-quality tool construction. Programmers can modify pieces of source code and immediately observe changed behavior in running programs. Graphic frameworks such as Morphic [24] leverage this idea for programs with interactive, visual output. Programmers can directly explore and adapt graphical objects and hence shape the user experience as desired.

With our data-driven perspective, we proposed a novel mechanism to further facilitate the idea of modifying the tools in use and applied it to a range of graphical tools for the programming domain. Given this, programmers can perceive the requirements for being both tool user and tool builder differently. We think that this perspective on graphical tools can inspire the creation of new trade-offs in modularity for both data-providing projects and interactive views.

Acknowledgments

We wish to thank Richard P. Gabriel, Tim Felgentreff, Tobias Pape, Stephanie Platz, Marko Röder, and Lauritz Thamsen for valuable feedback. We gratefully acknowledge the financial support of HPI's Research School[13] and the Hasso Plattner Design Thinking Research Program.[14]

References

[1] A. Bergel, S. Ducasse, C. Putney, and R. Wuyts. Creating Sophisticated Development Tools with OmniBrowser. *Elsevier Computer Languages, Systems & Structures*, 34(2):109–129, 2008.

[2] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as Objects in Newspeak. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*, pages 405–428. Springer, 2010.

[3] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola Jr. Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE) Volume 1*, pages 455–464. ACM/IEEE, 2010.

[4] P. Bunge. Scripting Browsers with Glamour. Master's thesis, University of Bern, 2009.

[5] V. Bykov. Hopscotch: Towards User Interface Composition. In *Proceedings of the 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT)*. Springer, 2008.

[6] A. Chis, T. Gîrba, and O. Nierstrasz. The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers. In *Proceedings of the 7th International Conference on Software Language Engineering (SLE)*, 2014.

[7] T. A. Corbi. Program Understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[8] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. Van Wijk, and A. Van Deursen. Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. In *Proceedings of the 15th International Conference on Program Comprehension (ICPC)*, pages 49–58. IEEE, 2007.

[9] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A Systematic Survey of Program Comprehension Through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[10] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers. The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse. In *Proceedings of the 9th International Conference*

[13] `www.hpi.uni-potsdam.de/research_school`

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

on Principles and Practice of Programming in Java (PPPJ), pages 71–80. ACM, 2011.

[11] K. De Volder. JQuery: A Generic Code Browser with a Declarative Configuration Language. In *Practical Aspects of Declarative Languages*, pages 88–102. Springer, 2006.

[12] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss. Debugger Canvas: Industrial Experience with the Code Bubbles Paradigm. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 1064–1073. ACM/IEEE, 2012.

[13] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature Location in Source Code: A Taxonomy and Survey. *Wiley Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[14] S. Hahn, J. Trümper, D. Moritz, and J. Döllner. Visualization of Varying Hierarchies by Stable Layout of Voronoi Treemaps. In *Proceedings of the 5th International Conference on Information Visualization Theory and Applications (IVAPP)*. SCITEPRESS - Science and Technology Publications, 2014.

[15] E. Hajiyev, M. Verbaere, and O. De Moor. Codequest: Scalable Source Code Queries with Datalog. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*, pages 2–27. Springer, 2006.

[16] R. Hirschfeld, M. Perscheid, and M. Haupt. Explicit Use-case Representation in Object-oriented Programming Languages. In *Proceedings of the 7th Symposium on Dynamic languages (DLS)*, pages 51–60. ACM, 2011.

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

[18] M. Kersten and G. C. Murphy. Using Task Context to Improve Programmer Productivity. In *Proceedings of the 14th International Symposium on Foundations of Software Engineering (FSE)*, pages 1–11. ACM, 2006.

[19] A. J. Ko, R. DeLine, and G. Venolia. Information Needs in Collocated Software Development Teams. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 344–353. ACM/IEEE, 2007.

[20] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 492–501. ACM/IEEE, 2006.

[21] J. Lincke and R. Hirschfeld. Scoping Changes in Self-supporting Development Environments Using Context-oriented Programming. In *Proceedings of the 4th International Workshop on Context-oriented Programming (COP)*. ACM, 2012.

[22] J. Lincke and R. Hirschfeld. User-evolvable Tools in the Web. In *Proceedings of the 9th International Symposium on Open Collaboration (OpenSym)*. ACM, 2013.

[23] J. Lincke, R. Krahn, D. Ingalls, and R. Hirschfeld. Lively Fabrik A Web-based End-user Programming Environment. In *Proceedings of the 7th International Conference on Creating, Connecting and Collaborating through Computing (C5)*, pages 11–19. IEEE, 2009.

[24] 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.

[25] K. Mens, B. Poll, and S. González. Using Intentional Source-Code Views to Aid Software Maintenance. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM)*, pages 169–178. IEEE, 2003.

[26] P. Naur. Programming as Theory Building. *Elsevier Microprocessing and Microprogramming*, 15(5):253–261, 1985.

[27] Y. Ohshima, A. Lunzer, B. Freudenberg, and T. Kaehler. KScript and KSWorld: A Time-aware and Mostly Declarative Language and Interactive GUI Framework. In *Proceedings of the International Symposium on New Ideas in Programming and Reflections on Software (Onward!)*, pages 117–134. ACM, 2013.

[28] F. Olivero, M. Lanza, M. D'Ambros, and R. Robbes. Enabling Program Comprehension through a Visual Object-focused Development Environment. In *Proceedings of the Symposium on Visual Languages and Human-centric Computing (VL/HCC)*, pages 127–134. IEEE, 2011.

[29] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, 1998.

[30] M. Perscheid, B. Steinert, R. Hirschfeld, F. Geller, and M. Haupt. Immediacy through Interactivity: Online Analysis of Run-time Behavior. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE)*, pages 77–86. IEEE, 2010.

[31] E. S. Raymond. *The Art of UNIX Programming*. Addison-Wesley Professional Computing Series, 2004.

[32] J. Sillito, G. C. Murphy, and K. De Volder. Asking and Answering Questions During a Programming Change Task. *IEEE Transactions on Software Engineering*, 34(4):434–451, July 2008.

[33] D. Spinellis. UNIX Tools as Visual Programming Components in a GUI-builder Environment. *Wiley Software: Practice and Experience*, 32(1):57–71, 2002.

[34] B. Steinert, M. Taeumel, J. Lincke, T. Pape, and R. Hirschfeld. CodeTalk: Conversations About Code. In *Proceedings of the 8th International Conference on Creating, Connecting and Collaborating through Computing (C5)*, pages 11–18. IEEE, 2010.

[35] M. Taeumel, T. Felgentreff, and R. Hirschfeld. Applying Data-driven Tool Development to Context-oriented Languages. In *Proceedings of 6th International Workshop on Context-oriented Programming (COP)*. ACM, 2014.

[36] C. Treude and M. Storey. Work Item Tagging: Communicating Concerns in Collaborative Software Development. *IEEE Transactions on Software Engineering*, 38(1):19–34, 2012.

[37] 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.

[38] M. Vaziri, O. Tardieu, R. Rabbah, P. Suter, and M. Hirzel. Stream Processing with a Spreadsheet. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*, pages 360–384. Springer, 2014.

A. The Introductory Example

We gave an introductory example for a script using Unix filters to extract URLs from a Wikipedia page. This is how a comparable script can look in our framework using only transformations but no rules to generate an intermediate model for widgets:

```
[:token | (WebClient httpGet:
  ('http://en.wikipedia.org/wiki/{1}' format: {token}))
    content lines].
[:line | 'href="/wiki/[^"]*"' asRegex matchesIn: line].
[:tokens | tokens asSet asOrderedCollection sorted].
[:url | (url copyReplaceTokens: 'href="'
             with: 'http://en.wikipedia.org')
               allButLast].
```

Notice the more data-driven appearance of adjacent Smalltalk blocks compared to a functional implementation that uses the collection interface (subsection 2.2). These four transformations, respectively scripts, are used in Figure 1 extracting `text` properties after each transformation like this:

```
[:object | #text -> object asString].
```

Obviously, it is the programmers' choice to which extent they employ this mechanism. A similar script with clearer dataflow semantics can look like this:

```
[:token | 'http://en.wikipedia.org/wiki/', token].
[:url | WebClient httpGet: url].
[:response | response content lines].
[:line | 'href="/wiki/[^"]*"' asRegex matchesIn: line].
[:tokens | tokens asSet asOrderedCollection sorted].
[:url | url copyReplaceTokens: 'href="'
            with: 'http://en.wikipedia.org'].
[:url | url allButLast].
```

Given a file handle, one can also use a script to write the results into a file:

```
[:url :file | file nextPutAll: url, String cr].
```

B. The Code Browser

Figure 4 and Figure 5 shows parts of a code browser that lists packages, classes, and methods. There are three scripts for that; the first transforms the package organizer into a list of packages:

```
[:packageOrganizer | packageOrganizer packages].
[:package | #text -> package name].
```

The second script transforms package into classes:

```
[:package | package classes].
[:class | #text -> class name].
```

The third one transform classes into methods:

```
[:class | class theNonMetaClass methodDict values,
          class theMetaClass methodDict values].
[:method | #text -> method selector].
```

C. The Commit Tool

The commit tool supports browsing changed packages with diffs, browsing repositories for their versions, and storing source code into a repository. Here, we list all source code that is necessary to describe such a tool as illustrated in Figure 6.

For the packages, we use a list-based widget. Given a token, list all matching packages:

```
[:token | PackageOrganizer default packages
  select: [:package |
    package name
      includesSubstring: token asString
      caseSensitive: false]].
[:package | #text -> package name].
```

For the repositories, we use a list-based widget, too. Given a package, list all registered repositories in the corresponding working copy:

```
[:package | package workingCopy repositoryGroup
  repositories].
[:repository | #text -> repository description].
```

For invoking the commit, we use a button. Given the modified package, its repository, and a commit message, configure the button so that a new version is stored on a click:

```
[:package :repository :message |
  package workingCopy in: [:wc | {
    #text -> 'Commit'.
    #clicked -> [[repository storeVersion: (wc
        newVersionWithName: wc uniqueVersionName
        message: message)]] }] ]
```

For the list of versions, we use a list-based widget. Given the repository and a particular package, list all versions:

```
[:repository :package | repository
  versionNamesForPackageNamed: package name].
[:version | #text -> version name]
```

For the list of changes, we use a list-based widget. Given the package and its repository, ask the working copy for change operations:

```
[:package :repository | (package workingCopy
  changesRelativeToRepository: repository)
    operations].
[:operation | #text -> operation summary].
```

Finally for the source code diff, we use a rich text box. Given the patch operation, show its code as formatted diff:

```
[:operation | #text -> operation sourceText].
```

D. Tools for Program Comprehension

In the exemplary case in section 4, we presented a programmer that wrote several scripts for exploring source code and run-time state as well as for talking to a colleague. The following test case failed and made the programmer struggle:

```
testTopLeftField
  "Tests, whether the playfield's topLeftField is correctly updated
  when it has been replaced."
```

```
  | pipe |
  pipe := PlBendPipe new.
  (playField fieldAtPoint: 0@0) replaceBy: pipe.
  self assert: (playField topLeftField = pipe).
```

While stepping through the code, the programmer discovered a check in the called `replaceBy:` method:

```
replaceBy: aPipe
  "Replaces itself by an aPipe and links the new item with its
  new neighbours as well as the new neighbours with the new
  item."
  self canBeReplaced ifFalse: [^ self].
  aPipe
  leftNeighbour: self leftNeighbour;
    rightNeighbour: self rightNeighbour;
    topNeighbour: self topNeighbour;
    bottomNeighbour: self bottomNeighbour.
  self changed: #score with: aPipe scoreEmpty.
```

For the call stack in the debugger, we use a list-based widget. Given the suspended process, extract all stack frames:

```
[:process | process suspendedContext stack].
[:frame | #text -> frame printString].
```

For the source code, we use a text box. Given a selected stack frame, show the source code and highlight the program counter:

```
[:frame | | range |
  range := frame debuggerMap
    rangeForPC: frame pc "program counter (PC)"
    contextIsActiveContext: false. "for PC interpretation"
  { #text -> (frame method getSource makeSelectorBold
      addAttribute: (TextColor color: Color blue)
      from: range first
      to: range last) }].
```

For the process control, we use a button bar. Given the suspended process, map buttons to some process actions:

```
[:process | {
  #text -> 'Rewind'.
  #clicked -> [[ process suspendedContext in: [:old |
    (process popTo: old sender) == old
      ifTrue: [process restartTop;
                      stepToSendOrReturn]] ]] }].
[:process | {
  #text -> 'Step Into'.
  #clicked -> [[ process step: process suspendedContext.
                  process stepToSendOrReturn. ]] }].
[:process | {
  #text -> 'Step Over'.
  #clicked -> [[ process suspendedContext in: [:old |
    process completeStep: old.
    process suspendedContext == old
      ifTrue: [process stepToSendOrReturn]] ]] }].
```

For arguments in the arguments browser, we used a list-based widget. Given the selected stack frame, parse the source code to show the argument names:

```
[:frame | RBParser parseMethod: frame method getSource].
[:node | node arguments].
[:node | #text -> node token value].
```

For the actual instances, we used also a list-based widget that arranges multiple items in each line. Given the selected stack frame *and* the selected argument node, collect run-time data and show results:

```
[:node :frame | (TraceManager default
  argumentObjectsForReference: frame method
  methodReference)
    select: [:assoc | assoc key = node token value]].
[:assoc | assoc value].
[:points | points asSet asOrderedCollection sorted].
[:point | #text -> point asString].
```

The programmer's chat tool requires an established Jabber connection. It filters the roaster by the authors of the selected method like this:

```
[:connection :frame |
  | authors map |
  map := Dictionary newFrom: {
    'ba' -> 'Bob'
    'dt' -> 'Dave'
    'mj' -> 'Max' }.
  authors := frame method versions
    collect: [:version | map at: version author].
  connection buddies select: [:buddy |
    authors includes: buddy nickName] ].
[:buddy | {
  #text -> buddy nickName.
  #icon -> (buddy status = 'online'
    ifTrue: [UiIcons bulletGreen]
    ifFalse: [UiIcons bulletWhite]) }].
```

For the chat history, we use a text box. The script is re-evaluated on incoming messages Given the selected buddy, show the previous messages:

```
[:buddy | { buddy.
            LocalChatHistory messagesFor: buddy }].
[:buddy :message |
  #text -> ('<font color="#AAAAAA">{1}</font> - <font
  color="#0000FF">{2}</font><br>{3}'
    format: {
      buddy nickName.
      (message at: #method) ifNil: [''] ifNotNil: [:m |
        m methodClass name, ' \>\> ', m selector].
      message at: #mesage}) asHtmlText].
```

For the send button, we use a button bar like in the debugger. Given the buddy, the stack frame, and the current message, configure the button:

```
[:buddy :frame :message |
  #text -> 'Send Message'.
  #clicked -> [[
    buddy send: (Dictionary newFrom: {
      #message -> message.
      #method -> frame method)]] ].
```