In Hasso Plattner, Christoph Meinel, and Larry Leifer (eds.)
Design Thinking Research: Taking Breakthrough Innovation Home (pages 161-182)
Springer 2016 (doi:10.1007/978-3-319-40382-3_11)

Making Examples Tangible: Tool Building for Program Comprehension

Marcel Taeumel and Robert Hirschfeld[1]

Abstract Best practices in design thinking suggest creating and working with tangible prototypes. In software engineering, programmers interact with source code more than with customers. Their intent is to understand the effects of abstract source code on programs in execution. Existing tools for program exploration, however, are tailored to general programming language concepts instead of domain-specific characteristics and programmer's system knowledge. In this chapter, we establish the need for adapting programming tools in use when *navigating*, *viewing*, and *collecting* examples to increase tangibility, that is, clarity of a concept or idea based on what can be experienced on screen. We present our Vivide tool-building environment, which is a data-driven, scriptable approach to constructing graphical tools with low effort. By exploring common programming scenarios, we conclude that tool building does not have to be a detached, effortful activity but can be accomplished by the same programmers who detect deficiencies during their programming tasks. Then exemplary information about software systems can become tangible.

1 Introduction

Best practices in design thinking include *prototyping*, which helps verify assumptions and gain a better understanding of the often abstract and unclear problem and solution space. Such prototyping activities will typically produce *tangible artifacts*. This means that designers work with real-world materials such as paper, glue, pencils, whiteboards, and index cards. Externalizing thoughts and ideas in simple but concrete things can foster team communication or enable first user testing. The quality of prototypes can range from low-end to high-end while retaining their *exemplary* nature: It's not about experiencing the final product but about holding in your hands a low-cost, incomplete, yet tangible analogy.

When the product is going to be a piece of software, prototyping can support programmers and customers to talk about requirements in a shared language. Typical tangible artifacts include user stories [1] on index cards, bricolages of graphical user interfaces [2], or descriptive personas [3] on whiteboards.

[1] Software Architecture Group
Hasso Plattner Institute, University of Potsdam, Germany
Email: firstname.lastname@hpi.de

Despite the many social aspects involved in software engineering, most of the time programmers have to focus on *talking to source code* instead of customers. Among all information related to software systems, only the source code is always up-to-date because it describes the system's actual behavior. Every programming activity includes reading and modifying source code. There is in fact an overwhelming amount of information available in large systems. Not only the numerous lines of source code but also related artifacts, such as program execution traces and external documentation, can support understanding. Programmers continuously ask questions about system parts while fixing bugs or adding features. The helpful answers to these questions represent *tangible examples* of information needed to accomplish programming tasks.

In programming-the notion of tangibility does not address primarily a physical representation but an aspect of cognition. Examples representing concepts, mechanisms, or intents should be "capable of being precisely identified or realized by the mind"[2] to be tangible. In a given task, programmers have to understand the particular mapping between the *problem domain* and the *source code* as well as between the source code and the software system in *execution*. General questions include: "How is domain knowledge represented in source code?" and "How are the rather abstract descriptions from the source code put into action during program execution?" as depicted in Figure 1.1.

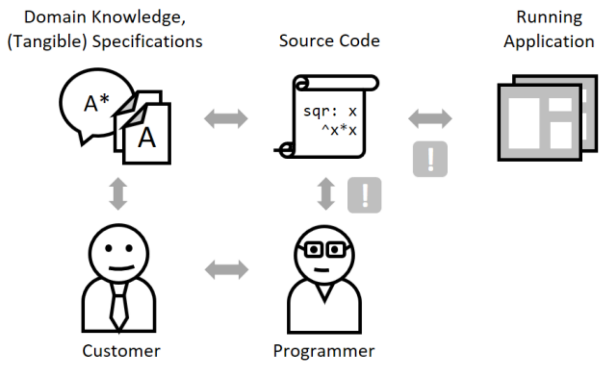


Fig. 1.1. Programmers write source code to make applications fit customer specifications. Large software systems pose challenges in uncovering the mapping between domain and code as well as code and running applications.

To acquire this understanding, programmers use *tools* for *collecting*, *navigating*, and *viewing* source code and other related software artifacts. These tools are called browsers, editors, debuggers, or explorers. Once created without anticipating specific domains, these tools provide *only generic support* for the underlying programming language constructs. For example, the language Smalltalk knows packages,

[2] Definition of "tangible" from http://merriam-webster.com, accessed on Dec 3, 2015

classes, categories, and methods. Tools support finding those artifacts and navigating their relationships. There are common strategies to approach a larger system such as “top-down” where programmers begin with the most abstract or coarse-grained structure and then dive into the details [4].

However, program comprehension is a creative activity influenced by human factors and domain-specific characteristics. There is typically no single strategy that works equally well for all programmers or in all known domains. On the one hand, people vary in terms of existing knowledge, cognitive capabilities, mood and motivational triggers [5], or even vision. On the other hand, domains carry specific terms and rules that have to be considered when using standard tools. For example, to answer “Why doesn’t this Tetromino[3] rotate clockwise when the arrow-right key is pressed?” requires mapping from specific terms to generic actions, which the tools dictate in this case as “Add breakpoint”, “Browse declaration”, and “Open documentation”.

By accommodating personal preferences and domain-specific characteristics, programmers are expected to find tangible examples more efficiently. Many tools offer a *means of configuration* by switching color schemes, keyboard shortcuts, window layouts, content filters, or integration with other tools. This possibility can reduce the number of user interactions and the chance to make mistakes due to cognitive overload or slip ups [6]. Hence, it can save time. As tools being software systems themselves, programmers have the skills to approach any task from the best possible angle.

However, there are two serious challenges that affect using these skills: (1) Not many programmers reflect regularly about their working habits yet come up with ideas for improvement and (2) simple tool customization is quite limited, extended customization typically not worth the usually high effort. At the end of the day, current habits remain unchanged. Programmers keep on using generic tools for a specific information space. This habit impedes navigating, viewing, and collecting software artifacts and hence finding the tangible examples that quickly answer program comprehension questions.

We think that programming languages and programming environments influence the way programmers think about tooling and the possibilities for improving their workings. In this chapter, we describe several existing means and triggers that foster the creative nature of program comprehension to gather the tangible examples more efficiently to answer the questions that arise in programming tasks. We emphasize the benefits of having a self-sustaining, reflective environment with access to the tool’s underlying source code. We present our tool building environment, called

[3] A “Tetromino” is the block in Tetris games. In such games the player has to arrange falling pieces of varying shapes to fill rows to gather points.

Vivide.[4] It is implemented in Squeak/Smalltalk[5] and employs a data-driven perspective on programming tools. It supports a scriptable way to create graphical tools for programming with low effort.

In section 2, we give background information and motivate the need for employing the best practices of self-sustaining programming environments such as Squeak/Smalltalk. We present several examples of deficiencies in generic programming tools in section 3, which reveal their deficiencies when applied to domain-specific programming tasks. We explain and apply our Vivide programming and tool building environment in section 4. Finally, we conclude our thoughts in section 5.

2 Learn about Your Environment's Possibilities

In this section, we establish the need for learning and applying the concepts that particular programming languages and environments provide. We argue that this is a requirement for efficient reflection about personal working habits and the adaptation of programming tools in use.

2.1 Being Aware of Different Concepts

Creativity draws from existing knowledge and previous experiences. Programmers can be creative when developing a program comprehension strategy, especially if they know and understand their surroundings, that is their programming languages, tools, and environments used. Programming tools are also described with source code, just like the application that has to be created for a customer. Hence, there is the chance that programmers can understand how tools work and how they can be modified to better support the circumstances. Unlike many users of software systems, programmers train particular skills that allow them to understand the building blocks of programs and their algorithmic, logic nature. Unfortunately, there are also many programmers who treat programming tools as "black boxes" and hence remain simply users. They are, however, unwilling to dig into internals and improve the modus operandi.

The tools' sources have to be available [7]. The curiosity of a programmer is not worth a dime, if there is no access to a human-readable description of the program. Source code gets translated into byte code to be interpreted by a virtual machine or compiled into machine code to be executed on the actual computer hardware. Such target representations are hardly readable by programmers. The preservation of the source code is required for maintenance. Having originated in a commercial context, many programming tools (or environments, respectively) such as Eclipse and Visual Studio do not offer sources. There are, however, full-source environments

[4] The Vivide environment: http://www.github.com/hpi-swa/vivide, accessed on Dec 3, 2015

[5] The Squeak/Smalltalk programming system: http://www.squeak.org, accessed on Dec 3, 2015

such as Squeak/Smalltalk where applications and tools are open, readable, and ready to be modified.

Whenever programmers learn new languages or tools, they build on existing knowledge and try to apply familiar concepts. This works quite well because many new ideas originate from previous experiences and retain best practices. For *imperative* programming languages, this might be the for-loop or if-else-conditional expression. For programming tools, this usually includes having text editors, copying code via the keyboard shortcut `[Ctrl]+[C]` and pasting contents via `[Ctrl]+[V]`, or setting breakpoints and invoking a debugger. However, programmers have to be open for new ideas. Especially when switching from a familiar language or environment to an unfamiliar one. That other environment might be used in familiar ways but its power can only unfold once its unique concepts become clear. For a brief example, in Squeak/Smalltalk, programmers can modify running applications by easily exchanging portions of code. However, it is also possible to kill and restart applications over and over again after every little modification. If programmers fail to learn and apply the concepts, patterns, and idioms, they cannot improve their working habits and hence only work inefficiently in the given environment.

2.2 Tool Mechanics

In this report, we focus on *graphical tools for programming*. These are tools that have windows, buttons, lists, text fields, or other kinds of interactive widgets. We think that programmers can benefit from graphics-based interfaces in terms of increased information density and convenient input methods such as mouse and touch. Text-based interfaces, for example command lines, are still popular in several communities and maybe one indication for inconvenient designs in the graphical world. However, this is precisely where programmers can take the opportunity to tailor their tools as needed. This can work if the mechanics of the underlying tool-building framework are comprehensive and easy to apply.

There are many ways to model the structure of programming tools. We think that it is useful to distinguish between the data that is accessed and the visuals that are produced as depicted in Figure 2.2.1. For tool builders, a *query language* is used to access the data. For tool users, a *presentation language* has to be learned to make sense of the visuals. Usually, there is also a *mapping language* because many software artifacts do not have an inherent graphical representation and hence have to be mapped to the graphical properties of standard widgets such as scrollable lists or text boxes.

Presentation languages encompass interactive, graphical widgets. For example, standard tools offer buttons, lists, trees, or tables. Sometimes they have maps or charts to visualize larger data sets and embed them into context in a meaningful way. This typically two-dimensional output is accompanied by mouse, keyboard, or touch input. There is usually a high degree of reuse of well-known concepts in new tools to support learning and foster best practices. For example, many tools have overlapping windows, tool bars, context/pop-up menus, save dialogs, or keyboard shortcuts. Depending on the domain, there might also be several unique widgets such as sheet editors in music composition tools [8].

Query languages support accessing and preparing data for widgets. They are programming languages with a specific focus. For example, if the information is stored in a relational database, SQL can be used to access those tables and to perform filter and aggregation operations. Given the concept of tables and rows, the expression "SELECT name FROM customers WHERE age > 60" reads the table "customers" and selects the rows with a certain age value to finally return the "name" column. One might easily have a textual or graphical representation in mind when querying data but such languages are independent of presentation. Tools can present the same information in different ways.

Mapping languages are required because the data providers and the interactive widgets usually do not speak a shared language, meaning that there is no inherent graphical representation for many software artifacts. Of course, it is easy to map information to textual representations because many data providers, internally, talk text-only. In information technology, textual representation of information is very important and is standardized, for example in the Unicode standard. This standardization is necessary for the sake of sharing, persistence, and long-lasting comprehension. However, in programmers' minds, some software artifacts have a more vivid appearance than others. Computer graphics employs lines, shapes, colors, or animation to make digital information almost tangible on screen. List widgets, for example, may benefit from "icons" or "color" but the data is only text. Here, mapping languages can be used to "materialize" concepts, that is, for example, mapping the string "red" to actual color information to be displayed on screen.

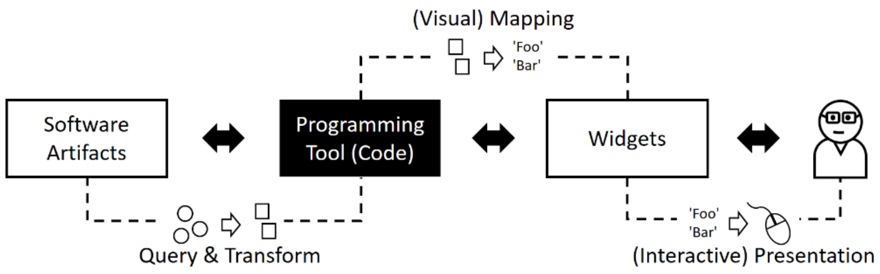


Fig. 2.2.1. Graphical tools for programming query system data to retrieve software artifacts such as source code, external documentation, and run-time traces. A subset of the artifacts' information is extracted and mapped to what interactive widgets support such as textual labels and color properties. The tool's source code is basically an adapter between databases and widgets.

2.3 Live Programming Environments

A central issue in program comprehension is how the rather abstract source code is put into action when it is executed on the computer. That means, understanding the correspondence between observable program output and its sources is paramount. To achieve this understanding, programmers execute smaller portions of a program called *tests* or pause program execution at certain points referring to (conditional) locations in source code, called *breakpoints*. Assumptions can be verified by exploring a program's run-time state or the result of a test run. Having such a unit of observation, small changes to the source code can also be used to check behavioral variations.

Edit-compile-run cycles can be minimized in live programming environments such as Squeak/Smalltalk. Programmers can evaluate any piece of source code in a text field within a particular context. There is always the global context, which means that `3 + 4` will evaluate to `7` and `Morph new openInHand` will create a blue rectangle attached to the mouse cursor as depicted in Figure 2.3.1. More specific contexts occur, for example, if an algorithm is in the middle of execution and it is paused by breakpoint. In that context, the keyword `self` evaluates to the object holding the shared state the algorithm is working with. For graphical objects, the expression `self color` will then evaluate to the object's current color. Although traditional environments such as Java/Eclipse[6] or C#/VisualStudio[7] do provide context when debugging, Smalltalk environments provide many more opportunities for programmers to work with run-time information. Setting breakpoints is then not always the first choice. Depending on the scenario, it can actually feel like "debugging mode is the only mode"[8], which is not feasible when writing, for example, a Java program.

Moreover, the Smalltalk programming language is also quite convenient to use as a query language for tool customization. The information is accessed in terms of Smalltalk objects such as morphs or colors as mentioned above. Program execution is structured with objects for processes, classes, methods, class instances, and method activations. Smalltalk can be used to query this information and prepare it for tools.

Smalltalk can be used as a mapping language, too. In Squeak, there is the interactive, graphical system called Morphic. All graphical objects are called *morphs*, which are basically rectangular areas that support composition in terms of holding sub-morphs. Although the boundaries blend, both the Smalltalk language and the Morphic system are important for mapping data to graphical representations. Programmers can describe custom morphs to display any kind of data. The Squeak

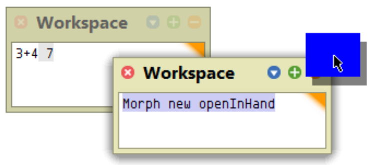
[6] The Eclipse programming environment, http://www.eclipse.org, accessed on Dec 3, 2015

[7] Microsoft VisualStudio programming environment, http://www.visualstudio.com, accessed on Dec 3, 2015

[8] "Debug Mode is the Only Mode", blog post from Gilad Brancha, http://gbracha.blogspot.de/2012/11/debug-mode-is-only-mode.html, accessed on Dec 3, 2015

environment provides tools that support code browsing, writing, and running activities with the help of windows, buttons, text fields, and lists. Every software artifact is an object, every graphical thing is a morph. This is a quite simple yet powerful concept that programmers can employ to accommodate any programming challenge.

The conceptual distance [9] between *morphs* and interactive *widgets,* such as buttons, is rather long., There are *tool building frameworks* that build on top of Morphic to minimize the amount of source code that has to be written for tools. We will explore our approach Vivide [10], which is a tool building environment with a data-centric focus, in the remainder of this chapter.



```
3+4 7
```

```
Morph new openInHand
```

Fig. 2.3.1. In Squeak/Smalltalk code can be evaluated in any text field. Here, two workspaces illustrate this concept. The blue rectangle is a morph and the result of the execution of the lower code snippet.

3 Reflect on Your Working Habits

In this section, we describe some prominent, recurrent scenarios where standard tools, which are aligned with programming language concepts, impede the discoverability of tangible examples and hence programming tasks. This should raise programmers' awareness to discover tool building as an opportunity to improve current program comprehension and modification strategies.

3.1 About Finding Tangible Examples

Programmers search for answers to program comprehension questions by navigating, viewing, and collecting software artifacts. Often, the means to navigate, view, or collect presents a challenge. Therefore, programmers should consider improving the situation by adapting the tools involved. A situation becomes challenging whenever programmers have to remember much information, interact with many tools back and forth in a loop, or continually ignore the same redundant or unimportant information over and over again. The screen real estate has to be optimized and necessary user input minimized. Finally, all important information has to be presented on screen so that the programmer can think about the current task with minimal cognitive overhead and come up with a solution involving where to modify the application to fix that bug or add that feature.

Programming tools are the means to navigate, view, collect, and even modify software artifacts. Primarily, programmers have to understand existing source code, modify existing source code, and write new source code. There is also other information that materializes in software artifacts. It originates from the operating system, programming language, execution environment, and other tools. Such artifacts are called files, classes, methods, tickets, emails, traces, processes and so on. They are typically related in one way or another. For example, emails can contain text referring to pieces of source code or traces can contain links to methods from program execution. Such relationships may not be explicit but have to be derived. Tools can help navigate relationships automatically to combine artifacts of different kinds with each other. For example, Mylyn [11] achieves this for source code and tasks, represented via tickets in an issue tracker. The tools' interactive widgets can help to display the information in a way that is helpful for the programmer to reveal news or recall what was already known and again of importance for the current task.

There are *facts* and there is *information* derived from those facts based on *rules*. For example, the birthdate of a person is a fact and its current age a derived information. In this respect, the source code comprises many facts. When running code, more data can be derived and interesting properties can be observed. Programmers use tools to learn about facts and also explore derived information. Some rules, however, are implicit and have to be inferred if necessary. For example, the way debuggers acquire access to the current program state is typically hidden in the internals of the debuggers' source code.

Programming environments support intra-tool communication. If the operating system is the programmers' environment, then files are typically used to store source code and exchange related artifacts between editors, compilers, or debuggers. There are programming environments that work on top of the operating system such as Emacs, Eclipse, or VisualStudio. Their means of tool communication enrich the file concept with, for example, text buffers or object-oriented structures. This simplifies the programming model for the tool builder. In Squeak/Smalltalk, programmers are almost completely shielded from the file system and only work in terms of objects, meaning classes, instances, methods, or method activations. While there is still support for text, object-orientation fosters abstract yet domain-specific thinking and also the creation of interactive, graphical programming tools. For example, if the project is about building an address book, then the objects might include persons or addresses and the tools can reflect their relationships with appropriate views and provide appropriate navigation links. A specialized object explorer, for example, might resemble a real-world address book to support program comprehension tasks.

The information space is changing. New source code gets written, and deprecated code gets removed. Yesterday's fact may have become derived from another fact based on some rules. Source code evolves in the way that it gets partially rewritten, called "refactoring." This occurs many times as programmers learn more about the respective problem domain and make changes accordingly. Such additional knowledge about the system or domain is also exchanged via emails or tickets. All kinds of software artifacts are constantly changing, which means that the tools will also have to change to accommodate domain-specific characteristics and personal traits.

3.2 Search for Examples and Navigate the Results

In most environments there is a text-based search tool that supports programmers in finding software artifacts by name or (text-based) contents by typing in a search term. Unfortunately, there is usually only a fraction of all information accessible, which is the source code and maybe some external documentation. However, this serves as a valid starting point for many program comprehension or modification tasks [12]. Programmers ask questions using terms from the problem domain and expect software artifacts to be named or described likewise in the code or comments. Having such starting points, exploration can continue with, for example, setting breakpoints and running the application.

An important tool for understanding the correspondence between source code and run-time is called "object explorer." In a class-based system such as Squeak, objects are instances of classes that have instance variables for object composition. This composite structure can be navigated because object explorers expose all such instance variables by name and with a textual summary of the particular referred to object, as depicted in Figure 3.2.1. Having this, the programming language dictates the functionality of this tool. Challenges arise when *domain-independent* relationships increase the tool interaction effort or when related artifacts have to be explored in separate tools. For example, morphs in Squeak encapsulate several properties, such as related to layouting, in an "extension" structure. Programmers always have to navigate this extension to find out about the current layout. When comparing the state of two different morphs, it is necessary to interact back and forth with two object explorers. However, simple integration points, for example the name of the instance variable, could be used to create a *combined* object explorer.

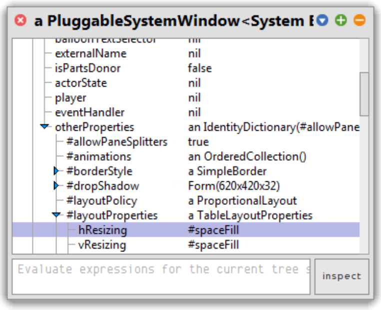

```
a PluggableSystemWindow<System E
  balloonTextSelector     nil
  externalName            nil
  isPartsDonor            false
  actorState              nil
  player                  nil
  eventHandler            nil
▼ otherProperties         an IdentityDictionary(#allowPane
    #allowPaneSplitters   true
    #animations           an OrderedCollection()
  ▶ #borderStyle          a SimpleBorder
  ▶ #dropShadow           Form(620x420x32)
    #layoutPolicy         a ProportionalLayout
  ▼ #layoutProperties     a TableLayoutProperties
      hResizing           #spaceFill
      vResizing           #spaceFill
Evaluate expressions for the current tree s    inspect
```

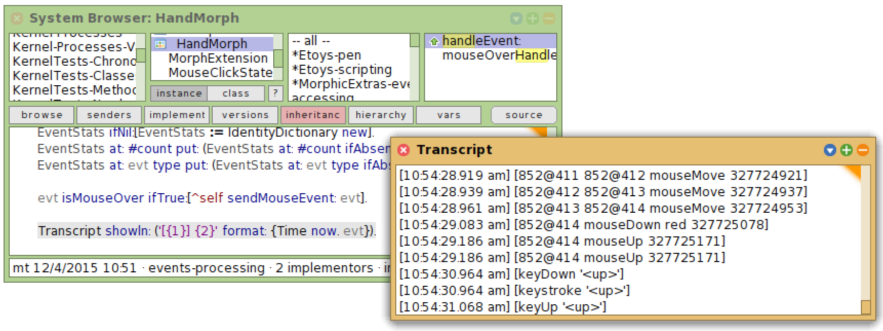
Fig. 3.2.1. The object explorer in Squeak. Exploring layout properties of a morph is challenging because morphs hide that state in an extension object and in an additional dictionary structure. In this example, the information is at the third level in the tree structure.

Refactoring tools require a set of source code artifacts to operate. For example, those tools support renaming or restructuring methods and update all related parts of the code. In dynamically typed programming languages such as Smalltalk, refactoring tasks benefit from additional information, such as run-time types and user-defined filters, to prevent inadvertent code changes. However, embedding a refactoring activity into an exploration activity can be challenging if tool integration is missing. All the different kinds of information involve handling separate tools and hence increase the cognitive effort, and also the risk of making mistakes. Sometimes, a rule that describes integration points can be simple like “Only consider the source code that I’ve modified during the last two hours.” Such a rule should be manifested in tools to optimize the current program comprehension strategy.

3.3 *View Information about Software Artifacts*

Programmers perceive software artifacts by viewing a subset of the artifacts’ information on screen. Mainly, there are inherent textual descriptions such as names or numerical values. However, there is usually more information about an artifact available than there is screen space and programmers only want to see the relevant details. Hence, filtering is a common way to customize the tools’ widgets. Many tools anticipate this action directly in the user interface without having to modify their source code.

The *console*, in Squeak called *Transcript*, is a common tool used by programmers for debugging. The practice is sometimes referred to as *printf-debugging* because standard libraries for the C programming language offer the function `printf` to write text to the standard console output. In Squeak, this corresponds to `Transcript show: someObject name`, which, for example, prints the object’s name. Programmers use the Transcript to trace information in the program without having to pause its execution. They have to map data or object structures to text but they can access anything from the particular context. Challenges arise when programmers fail to extract the relevant information and thus have to re-execute the presumably deterministic part of the program. As the output is typically in a text format, there is no other way to explore the underlying software artifacts with this strategy. See Figure 3.3.1 for an example.



```
EventStats ifNil:[EventStats := IdentityDictionary new].
EventStats at: #count put: (EventStats at: #count ifAbsen
EventStats at: evt type put: (EventStats at: evt type ifAb

evt isMouseOver ifTrue:[^self sendMouseEvent: evt].

Transcript showln: ('[{1}] {2}' format: {Time now. evt}).
```

mt 12/4/2015 10:51 · events-processing · 2 implementors · i

```
[10:54:28.919 am] [852@411 852@412 mouseMove 327724921]
[10:54:28.939 am] [852@412 852@413 mouseMove 327724937]
[10:54:28.961 am] [852@413 852@414 mouseMove 327724953]
[10:54:29.083 am] [852@414 mouseDown red 327725078]
[10:54:29.186 am] [852@414 mouseUp 327725171]
[10:54:29.186 am] [852@414 mouseUp 327725171]
[10:54:30.964 am] [keyDown '<up>']
[10:54:30.964 am] [keystroke '<up>']
[10:54:31.068 am] [keyUp '<up>']
```

Fig. 3.3.1. “printf debugging” in Squeak. If programmers do not extract helpful information, they have to modify debugging statements and re-execute the program.

There are many tools that require a textual representation of objects for on-screen display such as the object explorer in Figure 3.2.1. When there is a class hierarchy with a common base class—see `Object` in Squeak or Java—programmers can overwrite `#printString` (resp. `toString()`) in a subclass to accommodate domain-specific characteristics. The default textual representation of objects in Squeak is constructed with the class name and the identity hash like “aPerson(1234),” which is rather abstract. In fact, this example is not tangible at all. A slightly better mapping could be “John, Doe, 32.” However, the underlying concept of persons would be rather implicit and only discoverable if the programmer associates that information with the concepts of *name* and *age* and eventually with a *person*. A very elaborate version can reveal all details: “aPerson (forename: John, surname: Doe, age: 32).” However, this representation not scale when printed on the console among much other information. It is also independent from any particular programming task or personal preference or existing knowledge. There is a need to adapt the textual representation ad-hoc according to the current situation.

Many programming tools have list-like widgets. Examples include class browsers, search result explorers, and save dialogs. List-like widgets provide an overview and typically some interaction to view, select, move, or drag items. Although there is often support for filtering, lists usually lack support for shaping an item’s graphical appearance, including layout properties. See Figure 3.3.2 for the same artifacts displayed in a list, a tree, and a tree map. Note that it is not feasible to modify `#printString` as described above because programmers may want to see different information in different tools and, most importantly, according to the relevance for the current programming task.

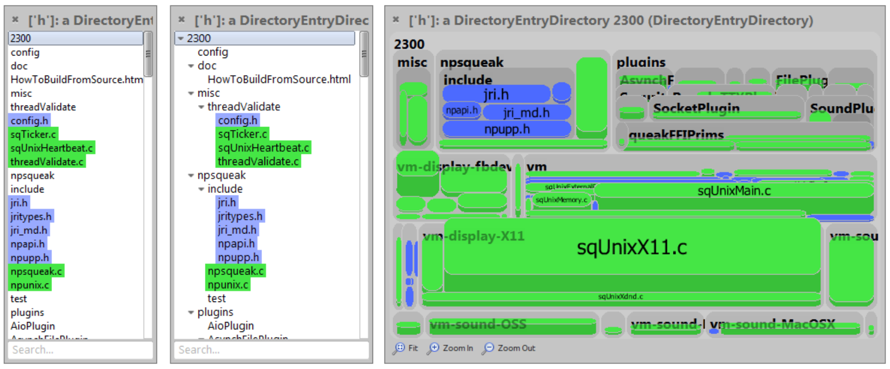
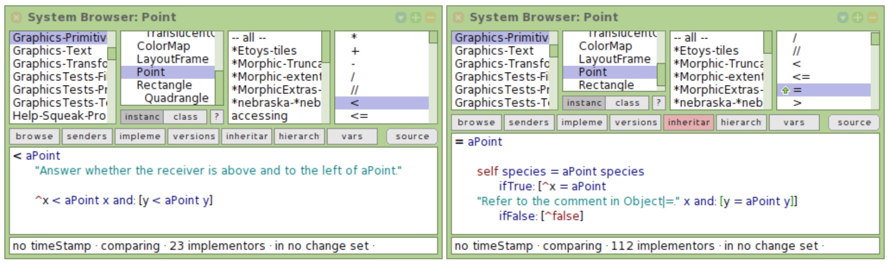


Fig. 3.3.2. Three views that display the same software artifacts but reveal different information. The list (left), the tree (middle), and the tree map (right) support labels and colors. The list ignores the hierarchical structure. The tree has much unused whitespace. The tree map has a space-filling approach.

3.4 *Collect Useful Pieces of Information*

Programmers can take notes in files via text editors, which are part of many programming environments. However, a textual representation of complex object structures requires filtering and summarization, which may be too early to do with the present system knowledge. Programmers may omit to write down important information. This practice suffers from the same problems as printf-debugging, as described above. In Squeak, there is a text-editor-like tool called *Workspace*, which supports source code evaluation (Figure 2.3.1). It also supports dropping graphical objects, which then get captured and are referenceable with a variable like `droppedMorph`. These objects can be explored by evaluating the snippet `droppedMorph explore` within that workspace. It can be beneficial to keep track of interesting software artifacts and defer setting up a representation on screen until later in the course of the programming task when concepts become clearer and examples more tangible.

Tools consist of one or more *windows* to manage their contents. Basically, windows are rectangular areas that show a document or a scrollable portion of it. Tool operations become accessible via push buttons, menu bars, and other additional widgets, typically arranged around a central view. Finding an efficient window manager for programming environments with graphical user interfaces has a long history [13]. Semi-automatic window layout strategies range from overlapping (Squeak) to tiling (Eclipse) to stacking (Web browsers) – or any combination. Managing the position and extent of tool windows is still part of programmers' frequent activities, and is in the focus of research [14].



```
< aPoint
    "Answer whether the receiver is above and to the left of aPoint."

    ^x < aPoint x and: [y < aPoint y]
```

no timeStamp · comparing · 23 implementors · in no change set ·

```
= aPoint

    self species = aPoint species
        ifTrue: [^x = aPoint
    "Refer to the comment in Object|=." x and: [y = aPoint y]]
        ifFalse: [^false]
```

no timeStamp · comparing · 112 implementors · in no change set ·

Fig. 3.4.1. There is much redundant information visible when comparing, for example, two methods from the same class (here: `<` and `=` from `Point`). This is not necessarily a problem of window management but of the tool's user interface design.

While windows can be used to collect and arrange information about software artifacts on screen, their tool-driven characteristics entail much redundant and task-independent information. A window does not correspond to a single software artifact but many. Tool windows in Squeak, for example, are self-contained and display full context information such as code browsers do for displaying *class* definitions and *method* source code. This has the disadvantage that two code browsers that browse two methods from the same class will both display the same context information. This means both will display irrelevant message names, message categories, and other classes in the same package, and class categories with similar spelling (Figure 3.4.1). In Eclipse, tools use a tiling layout strategy and display more specific and less redundant information in their windows such as the class outline at the side and the dedicated text editor in the center. Tiling, however, is much more restrictive when it comes to collecting examples because screen space is limited and quickly exhausted if you can only arrange windows side-by-side. If you stack them, like tabbing in web browsers, you will save screen space but also sacrifice visibility of information completely. Programmers should be able to choose the best way to arrange examples on screen because this level of control can improve tangibility.

3.5 Improve the Means to Navigate, View, and Collect

Tool modification is necessary to accommodate domain-specific tasks and individual system knowledge. A dedicated tool builder who usually creates tools with a *high level of prospective reuse* can only provide means for common tasks, which are usually aligned to the particular programming language and execution environment. Domain-specific characteristics and personal traits cannot be known upfront. Programmer who have concrete tasks, such as fixing bugs and adding features, are best suited to improve working habits by adapting the tools in use themselves.

Thus, whenever navigation repeatedly requires many user interactions, the visual display hides relevant details, or the collection of insights is challenging, programmers can take the chance to act as tool builders. As one would build a level editor for a game to support the creation of game content, programmers can build customized tools to support the work on any software system.

4 Apply Data-driven Tool Building

Programmers can find tangible examples for comprehension tasks by using programming tools for navigating, viewing, and collecting software artifacts. Such tools are built with a query language to access and process artifacts, a mapping language to cope with inappropriate or missing graphical representations, and a presentation language to serve the user with an interactive front-end. Whenever detecting deficiencies in tools, programmers have the skills to improve those tools ad-hoc – even for scenarios that may be unique.

However, tool building frameworks require a high effort for *tracing* a tool's observable deficiency to the responsible portion of the tool's source code. Additionally, code *changes* are rather verbose and tools do not *update* consistently so that programmers have to restart and hence constantly repeat certain interactions before proceeding with the actual programming task. Eventually, the prospective cost-benefit ratio might render such a tool adaptation pointless and hence programmers keep on using standard tools and inconvenient working habits.

We created a new tool building framework, called Vivide [10], that projects a data-driven perspective on graphical tools and employs a scriptable way to modify the tools in use with low effort. The extensible presentation language consists of common widgets such as buttons, text boxes, and lists. Both query and mapping language are based on Smalltalk and integrate seamlessly existing object-oriented code. In this section, we will describe Vivide's concepts in detail and apply them for *navigating, viewing, and collecting* software artifacts to increase tangibility of the exemplary information shown on screen.

4.1 The Vivide Tool Building Environment

Vivide [10] is a programming environment that supports programmers to focus on their domain-specific data. It projects a data-driven perspective on graphical tools and employs scripts that express rules for transforming data and extracting relevant information to be stored in a model. That model will be interfaced from interactive widgets such as lists or buttons. We think that by putting the domain-specific data (resp. software artifacts) in the foreground the notion of tools fades into the background. Programmers are more likely to create or modify tools as an unnoticed side-activity while navigating, viewing, and collecting relevant information.

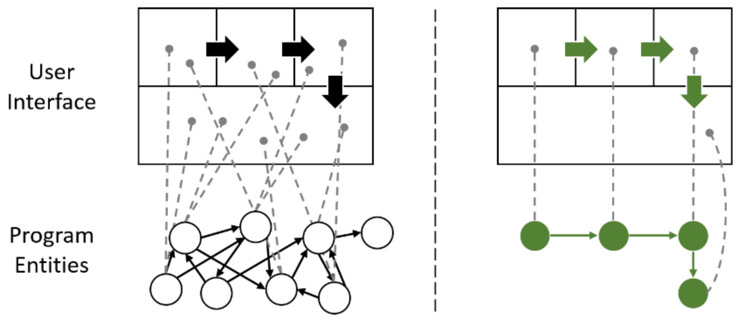


Fig. 4.1.1. Traditional tools (left) provide a rather complex, indirect correspondence between user interface and program entities; Vivide tools (right) provide a direct correspondence. Rectangular portions of the user interface are called *panes*. Programmers have to think about tools being only data-exchanging panes.

The Vivide environment provides a direct correspondence between all graphical parts of the user interface and the internal tool logic as depicted in Figure 4.1.1. Vivide is implemented in Squeak/Smalltalk and builds on top of the Morphic framework, which supports direct manipulation of all graphical objects. Every morph has a meta-menu, called *halo*. It can be invoked with a dedicated user interaction such as a click on the middle mouse button, that represents a graphical meta-interface to perform inspection and modification tasks. Vivide makes use of the halo concept to provide access to the underlying tool mechanics as depicted in Figure 4.1.2. Now, programmers can easily find responsible data transformation scripts starting with a visual impression and express modifications in the script source code. Due to this simple yet powerful abstraction, the Vivide framework can update all running tools consistently when scripts change. The programmer can continue the programming task without having to repeat previous tool interactions.

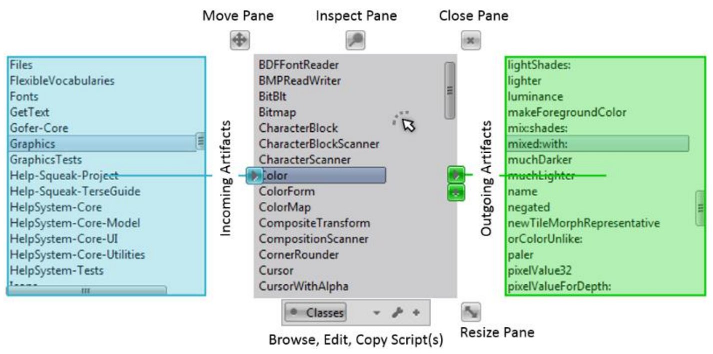


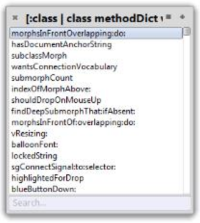
Fig. 4.1.2. Every graphical part (pane) of tools in Vivide has a halo that provides access to data-flow properties and the underlying script via floating buttons.

Thus, Vivide is not only a programming environment but also a tool building framework. Building tools means composing widgets, like in a GUI builder, and writing script code. Script code describes rules for transforming software artifacts and extracting relevant information to be used by widgets. For example, a script that transforms classes into methods and extracts the selector from methods to be displayed in a list can be created with only a few lines of code as depicted in Figure 4.1.3.

There is also a *wizard* involved that tries to detect 1:n, n:1, or n:m transformations to further reduce programming effort. In general, programmers have full control over the input and output of objects in a script. In Smalltalk terms, a script is a collection of *blocks* in the form `[:in :out | ]`, and the Vivide framework will initiate block evaluation with actual objects. The wizard expands `[:a | a + 1]` to `[:in :out | out addAll: (in collect: [:a | a + 1])]` and other expressions to similar constructs.

We think that programmers who build tools with Vivide will benefit from time-related advantages compared to traditional tool building approaches. This may have an impact on the cost-value ratio of tools and thus also on the whole tool building community.

```
{  [:class | class methods]
      -> { #view -> ListView }.
   [:method | { #text -> method selector }].
} openScriptWith: {Morph}.
```



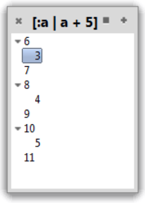
```
[:class | class methodDict
morphsInFrontOverlapping:do:
hasDocumentAnchorString
subclassMorph
wantsConnectionVocabulary
submorphCount
indexOfMorphAbove:
shouldDropOnMouseUp
findDeepSubmorphThat:ifAbsent:
morphsInFrontOf:overlapping:do:
vResizing:
balloonFont:
lockedString
sgConnectSignal:to:selector:
highlightedForDrop
blueButtonDown:
Search...
```

Fig. 4.1.3. In Vivide, only a few lines of Smalltalk code (left) are needed to describe an interactive tool (right). Here the tool operates on the `Morph` class, transforms it into methods, and displays the methods' selectors in a list that supports drag-and-drop for continuing the exploration.

4.2 Support Navigation with Adapted Tree Structure

Scripts in Vivide support describing tree structures to be used by widgets. Each script is a set of object transformation and property extraction rules. With this, programmers can transform any collection of input objects into any other collection of output objects. For example, programmers can navigate *existing relationships* or perform a computation to *derive new information*. The empty script is `[:in :out | out addAll: in]`, which just forwards all objects from the input buffer to the output buffer. After transforming objects, programmers can describe properties of interest such as `#text` as extracted in the example above. Alternating transformation and extraction means describing multiple levels of a tree structure. While plain list widgets might not take notice of such a tree structure, tree widgets will do as depicted in Figure 4.2.1. Programmers can adapt tree structures to simplify navigation and make efficient use of screen real estate. An example of this is presented in Figure 4.2.2.

```
{  [:a | a + 5].
   [:a | #text -> a].
   [:b | b even ifTrue: [b / 2]].
   [:b | #text -> b]
} openScriptWith: #(1 2 3 4 5 6).
```

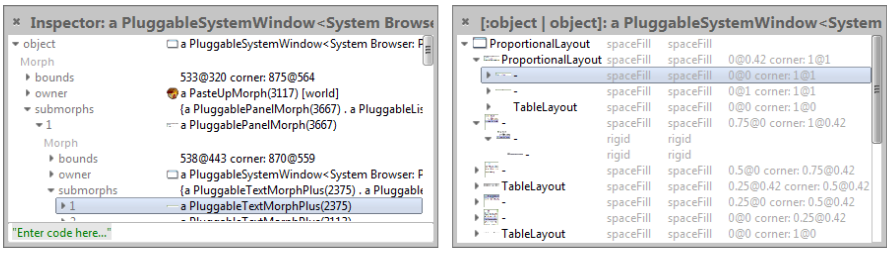


```
[:a | a + 5]
▾ 6
    3
  7
▾ 8
    4
  9
▾ 10
    5
  11
```

Fig. 4.2.1. Alternating transformation and extraction describes different levels (a and b) in the tree structure. Note that Smalltalk code can be used in all scripts. The wizard will expand all four blocks into the form `[:in :out | ...]`.

Scripts can process multiple data sources as a means of combination and integration. Multiple sources, meaning collections of objects, can be combined as the *Cartesian product.* The scripts still operate on collections of objects but these then contain *n-tuples*, where n depends on the number of data sources. For example, having two sources (1, 2) and (a, b, c), the script then handles ((1, a), (1, b), (1, c), (2, a), (2, b), (2, c)). Programmers have to know about this in their scripts. The aforementioned script wizard helps combine multiple data sources.

Programmers can then use Vivide scripts to describe arbitrary navigation paths via tree structures. A single tool can shorten navigation paths by only exploiting the relationships of interest as depicted in Figure 4.2.2.



```
Inspector: a PluggableSystemWindow<System Browse
▾ object          a PluggableSystemWindow<System Browser: P
  Morph
  ▸ bounds        533@320 corner: 875@564
  ▸ owner         a PasteUpMorph(3117) [world]
  ▾ submorphs     {a PluggablePanelMorph(3667) . a PluggableLis
    ▾ 1           a PluggablePanelMorph(3667)
      Morph
      ▸ bounds    538@443 corner: 870@559
      ▸ owner     a PluggableSystemWindow<System Browser: P
      ▾ submorphs {a PluggableTextMorphPlus(2375) . a Pluggable
        ▸ 1       a PluggableTextMorphPlus(2375)
"Enter code here..."
```

```
[:object | object]: a PluggableSystemWindow<System
▾ ProportionalLayout    spaceFill  spaceFill
  ▾ ProportionalLayout  spaceFill  spaceFill  0@0.42 corner: 1@1
    ▸ -                 spaceFill  spaceFill  0@0 corner: 1@1
    ▸ -                 spaceFill  spaceFill  0@1 corner: 1@1
    ▸ TableLayout       spaceFill  spaceFill  0@0 corner: 1@0
  ▾ -                   spaceFill  spaceFill  0.75@0 corner: 1@0.42
    ▾ -                 rigid      rigid
        -               rigid      rigid
  ▸ -                   spaceFill  spaceFill  0.5@0 corner: 0.75@0.42
  ▸ TableLayout         spaceFill  spaceFill  0.25@0.42 corner: 0.5@0.42
  ▸ -                   spaceFill  spaceFill  0.25@0 corner: 0.5@0.42
  ▸ -                   spaceFill  spaceFill  0@0 corner: 0.25@0.42
  ▸ TableLayout         spaceFill  spaceFill  0@0 corner: 1@0
```

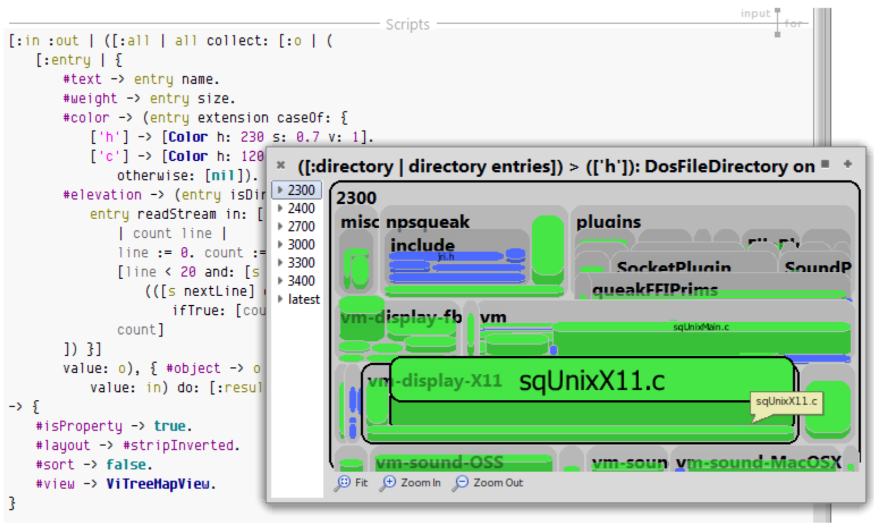
Fig. 4.2.2. On the left there is a default object explorer showing details about the window of a code browser. On the right the same object structure is simplified by only navigating the composite GUI structure (sub-morphs) while exposing layout properties. The same object, a text input field, is selected in both windows to emphasize the efficient use of available screen space.

4.3 *Support Views with Named Properties*

Scripts can be used to map any information of a software artifact to something that widgets can use. For example, programmers can derive color information, setup labels or tooltips, and so on. Such *property extractions* have the form of an array with associations as depicted in Figure 4.3.1. Besides visual mappings, any *object-specific* information can be provided for widgets given a name. Common properties include `#text`, `#icon`, `#balloonText`, and `#color`. The widgets decide about those means of configuration and can, theoretically, adapt their whole behavior. It is not part of the concept of Vivide to prescribe the use of such properties. The tree map in Figure 4.3.1 is able to use text, weight, color, and elevation.

Scripts also have a set of *object-independent* properties. For example, Vivide stores a flag `#isProperty` to distinguish between object transformation and property extraction scripts. Widgets also get the chance to configure themselves according to script-properties. The tree map in Figure 4.3.1 reads `#layout` and `#sort` to adapt its general layout strategy and sort order.

Programmers are in control of the presentation of software artifacts. They can choose between a set of views such as lists, tables, trees, tree maps, and other charts. Furthermore, they can tweak those views at the script level. Any change in script code will immediately update the corresponding views in the programming environment. Such short feedback loops support programmers in exploring information and finding examples. Those examples can become tangible if programmers can find an appropriate way to show them on the screen – tailored to the domain and personal preferences.



```
[:in :out | ([:all | all collect: [:o | (
   [:entry | {
      #text -> entry name.
      #weight -> entry size.
      #color -> (entry extension caseOf: {
         ['h'] -> [Color h: 230 s: 0.7 v: 1].
         ['c'] -> [Color h: 120
            otherwise: [nil]).
      #elevation -> (entry isDir
         entry readStream in: [
            | count line |
            line := 0. count :=
            [line < 20 and: [s
               (([s nextLine]
                  ifTrue: [cou
            count]
      ]) }]
      value: o), { #object -> o
         value: in) do: [:resul
-> {
   #isProperty -> true.
   #layout -> #stripInverted.
   #sort -> false.
   #view -> ViTreeMapView.
}
```

Fig. 4.3.1. A rather complex example of a script (background) that extracts properties for a tree map (foreground). Object-specific properties include text, weight, color, and elevation. View-specific properties include layout strategy and sort order. Left of the tree map, the tool also offers a tree representing part of a directory structure with source code files.

4.4 Support Collection with Arbitrary Containers

Vivide supports overlapping windows for tools like Squeak does. However, Vivide entails the idea that “window management” can be pluggable. Based on the authors’ experiences, overlapping windows suffice most of the time. Common tool windows, however, have their contents tiled or stacked as depicted in Figure 4.3.1. Within those tiles, the layout strategy may be different. List widgets, for example, arrange their items vertically side-by-side. In another example, tree maps can have overlapping items when supporting elevation – or even a 3D canvas. Based on these observations, we think that programmers should be able to decide about the layout strategy for each level in a tool’s graphical hierarchy. The concept of rectangular tool building blocks, called *panes*, in Vivide is expanded to *multi-pane widgets*. These widgets encapsulate multiple panes to apply any possible layout strategy. For an example, see Figure 4.4.1. Such means of content organization are independent of actual visualizations. Programmers can employ them as required to make software artifacts more tangible on screen.

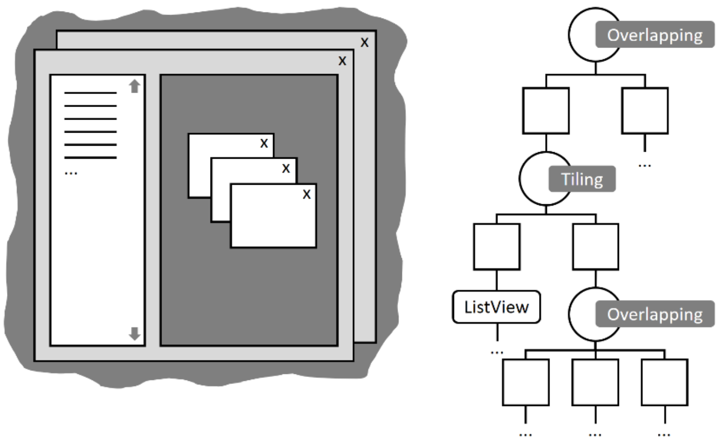
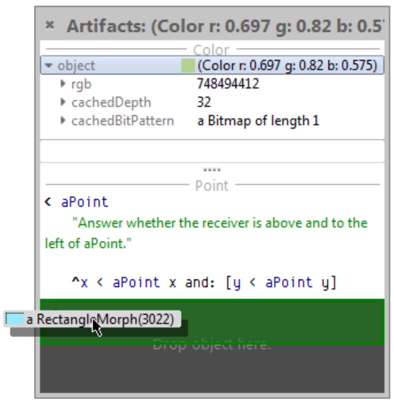


Fig. 4.4.1. Window management should be pluggable and any layout strategy applicable in any part of the graphical hierarchy as needed. Here, the hierarchy on the right models the tools on the left. In the hierarchy, circles denote *multi-pane widgets*, squares denote *panes*, and rounded squares other *morphs* in the world.

Collecting objects via drag-and-drop using a pointing device (mouse or touch) is a common practice in Vivide. Many widgets support dragging their objects. Then, there is are generic container to collect those objects as depicted in Figure 4.4.2. Programmers can collect pieces of source code that are otherwise scattered in the class hierarchy and view them side-by-side. Programmers can also mix different kinds of objects such as run-time artifacts and code artifacts. Although Vivide provides support at the level of graphical, interactive tooling, the Squeak/Smalltalk environment is the reason for run-time information being omnipresent in general.



object	(Color r: 0.697 g: 0.82 b: 0.575)
rgb	748494412
cachedDepth	32
cachedBitPattern	a Bitmap of length 1

```
< aPoint
    "Answer whether the receiver is above and to the
left of aPoint."

    ^x < aPoint x and: [y < aPoint y]
```

Fig. 4.4.2. Vivide provides a generic container to collect software artifacts via drag-and-drop. Each dropped artifact is displayed in an interactive view. Here, there is an object explorer for a color object, a code editor for a Point method, and a mouse cursor about to drop a rectangle morph.

5 Conclusion

Program comprehension benefits from tangible examples. Due to the abstract characteristics of source code, tangibility does not refer to a physical representation but to conceptual clarity and understanding. Regarding a concept in the source code, its correspondence in the problem domain has to be understood as well as its effects during program execution. Having this, finding the subset of relevant information and presenting them in a tangible way is a matter of efficient programming tools and programming environments. This efficiency is specific to the domain and to the programmer's personal preferences and existing knowledge. Traditional programming tools, however, align with generic programming language concepts; specific scenarios are not well supported. Tool adaptation seems beneficial but is typically expensive.

We presented the Vivide programming and tool building environment, which is a data-driven, scriptable, interactive approach to construct graphical tools for programming. We applied Vivide in several examples to illustrate ways to improve the means to navigate, view, and collect software artifacts. While programmers do not have to come up with the perfect solution right from the beginning, Vivide's direct feedback after each tool modification fosters an iterative and explorative working mode. Programmers can safely try out ideas and undo mistakes with ease. Even unique scenarios can be improved, and reuse can then become of secondary interest.

References

[1] M. Cohn, User Stories Applied, Pearson Education, Inc., 2004.

[2] B. Shneiderman and C. Plaisant, Designing the User Interfaces: Strategies for Effective Humand-Computer Interaction, Addison-Wesley, 2010.

[3] C. Courage and K. Baxter, Understanding Your Users: A practical guide to user requirements, Elsevier, 2005.

[4] A. Von Mayrhauser and A. M. Vans, "Program Comprehension during Software Maintenance and Evolution," *IEEE Computer,* vol. 28, no. 8, pp. 44-55, 1995.

[5] M. Csikszentmihalyi, Flow: The Psychology of Optimal Experience, Harper Perennial Modern Classics, 2008.

[6] D. A. Norman, The Design of Everyday Things, Basic Books, 2002.

[7] M. Weiser, "Source Code," *IEEE Comptuer,* November 1987.

[8] J. Wright, D. Oppenheim, D. Jameson, D. Pazel and R. Fuhrer, "CyberBand: A 'Hands On' Music Composition Program," in *Proceedings of the International Computer Music Conference*, 1997.

[9] E. L. Hutchins, J. D. Hollan and D. A. Norman, "Direct Manipulation Interfaces," *Human-Computer Interaction,* vol. 1, no. 4, pp. 311-338, 1985.

[10] 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*, 2014.

[11] 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)*, 2006.

[12] J. Sillito, G. C. Murphy and K. De Volder, "Asking and Answering Questions During a Programming Change Task," *IEEE Transactions on Software Engineering,* vol. 34, no. 4, pp. 434-451, 2008.

[13] B. A. Myers, "A Taxonomy of Window Manager User Interfaces," *IEEE Computer Graphics and Applications,* vol. 8, no. 5, pp. 65-84, 1988.

[14] D. Röthlisberger, O. Nierstrasz and S. Ducasse, "Autumn Leaves: Curing the Window Plague in IDEs," in *Proceedings of the 16th Working Conference on Reverse Engineering*, 2009.