In Christoph Meinel and Larry Leifer (eds.).
Design Thinking Research: Translation, Prototyping, and Measurement (pages 127-150)
Springer 2021 (doi:10.1007/978-3-030-76324-4_7)

Toward Patterns of Exploratory Programming Practice

Marcel Taeumel and Patrick Rein and Robert Hirschfeld

Abstract Patterns document best practices in many domains. For a long time, practitioners in the field of software engineering have been collecting and using such patterns too, to approach recurring design challenges. However, the challenges of efficient problem understanding and solution revising have no such form for efficiently communicating programming practice. It takes a long time to discover and learn such exploratory skills when using programming tools as is, without thorough reflection. We want to apply the idea of patterns to capture traditional and modern practices of exploratory programming. In this chapter, we begin to draft a pattern language, starting with four patterns to enable and control exploration, which we extracted from personal programming practice and experience.

1 Introduction

Software development often has the characteristics of a wicked problem [4]. Creating "good" software requires fulfilling external requirements relevant to the users and internal requirements relevant for the developers of the system. Users value easy-to-learn interfaces and useful features; developers appreciate code that can be understood by others and architectural designs that can be adapted easily. Such mediation between two sometimes quite different sides entails constant communication efforts as depicted in Figure 1. Often, we discover some of these (external and inter-

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

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

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

nal) requirements *only after* we build a version of the software. Given the complexity of many software systems, such timing can be fatal: we will only understand the full consequences at a point where systems become very difficult to change.

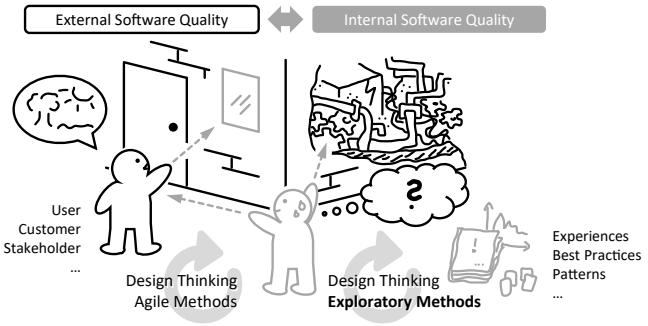


Fig. 1 Software developers mediate between user requirements and technical implementation. The goal is to master the implementation side and find a program design that can quickly adapt to changing user requirements.

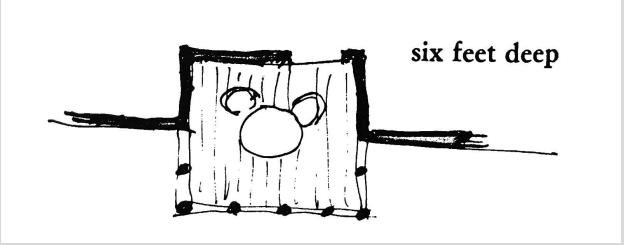
To tackle such "wickedness," some programmers engage in an intensive form of exploring the problem and solution space. When reasoning about their designs, they follow a programming style called "exploratory programming" [22, 27]. In this style, programmers do not try to construct the perfect solution right away, but aim to deeply understand the problem at hand, including possible solutions. They achieve such immersion by creating various prototypical implementations [27, 13], which they can directly try out and refine to eventually look at the system from many different perspectives. Thereby, the exploration process consists of many small, yet insightful, experimental changes to the prototype, whose consequences programmers can directly observe and learn from [27].

However, the actual skill set around exploratory programming is still difficult to learn. Proficient programmers typically invest years to acquire a number of best practices to support exploration: How to keep the overhead of iteration low? How to avoid breaking the system? How to get detailed feedback for answering any particular question? —Experienced programmers do not only know the steps to be taken, but also which ones work best in a particular situation, and how to adapt them to new ones. This knowledge is the result of years of practice or direct observation of other programmers during their explorations. There is no form to efficiently pass on this specific knowledge to the next generation of programmers.

We want to ease the learning of exploratory programming style by uncovering and documenting best practices as *patterns*. Patterns are a concise form of communicating the core of a solution obtained through experience [2, 1]. Patterns typically describe the *problem* they are tackling, the *context* in which they are to be applied, the core of the *solution*, and its *consequences*. Depending on a pattern's subject (or domain), some of these parts are left out or described in more detail. A common property of all pattern collections is their *generative* nature. By adapting the essential

part of the solution, they can be used to generate solutions that are tailored to new, unforeseen challenges.—We think that the pattern form is suitable to describe exploratory programming practices. Their structure can provide guidance to help us describe all relevant aspects of such practices, while at the same time being flexible enough to cover a variety of practices.

Where does the pattern form originate? The first-ever published pattern collection describes solutions for creating and shaping the living environment, including towns, houses, and individual rooms [2]. One such pattern is "six foot balcony." This pattern first describes the *context* in which the pattern is to be used: It then goes on to describe the main problem it tackles: "Balconies and porches which are less than six feet deep are hardly ever used." The *problem* is discussed further by describing the observation of how people make different use of balconies that are narrow and ones that are deep. The pattern then discusses the *solution* including variations such as enclosing balconies or recessing them into buildings:



"Whenever you build a balcony, a porch, a gallery, or a terrace always make it at least six feet deep. If possible, recess at least part of it into the building so that it is not cantilevered out and separated from the building by a simple line, and enclose it partially." —Christopher Alexander [2, p. 784]

The patterns in our collection are based on the experience of our research group. This experience is the result of more than a decade of research and education around exploratory programming systems. For both aims, we employ two exploratory programming environments: Squeak/Smalltalk [11, 7] and Lively [10, 14, 15]. In research, we create tools to support exploration of software designs by allowing programmers to change running systems, quickly adapt their tools to the task at hand, and gain more insights into the actual behavior of their system [15, 24, 19]. In education, we conduct lectures on software architecture and software engineering, as well as other courses on software design, end-user programming, and tool building. All these courses include practical work in one of these environments. This provides us with ample opportunity to observe the typical struggles of beginners in such environments.

This chapter presents a first draft of four relevant patterns on exploratory programming, which we observed in object-oriented system programming. Our contribution is as follows:

- "Tangible Names" and "Tangible Pixels"—Two patterns to *enable* exploration, which clarify the textual and graphical aspects of environments as entry points.
- "Configurable Constraints" and "Reliable Recovery"—Two patterns to *control* exploration, which clarify the aspect of trust in the environment as to avoid getting lost or wreaking havoc.

In section 2, we provide more detail on how and where we found the patterns for this chapter, that is, our experience with teaching and research in object-oriented programming systems. The following two sections describe the patterns: section 3 for *enabling* exploration and section 4 for *controlling* exploration. We close those descriptions with a reflective discussion on their maturity in section 5 as this is the first iteration of a pattern collection for exploratory programming. We conclude our thoughts in section 6.

2 From Experience to Pattern Form

We precede the presentation of this chapter's patterns on exploratory programming practice with more background on our expertise in object-oriented programming systems. At the end, we sketch the actual pattern form that we use, which deviates from existing pattern catalogs but fits our needs for capturing those practices.

2.1 Our Programming Experiences

The patterns described in this chapter result from our own experience with exploratory programming. To provide you with a background for the patterns, we will briefly outline our experience. We are a research group of 14 people focusing on programming tools and experience. Our experience with exploratory programming stems from engaging in it ourselves, from research around corresponding tools and environments, and from teaching undergraduate and graduate courses in exploratory environments.

Our research revolves around exploratory and live programming [21]. As part of this research, we design and create new tools and programming environments, designed to support specific exploratory practices. Many of the ideas and environments we build upon originated from the Learning Research Group at Xerox PARC [12]. The systems and ideas, which have stood the test of time, form the foundation for many of the following projects.

Exploratory programming entails new situations that are not supported by existing tools. The Vivide environment [26, 25, 24] supports exploratory programmers

in creating new tools or adapting their tools quickly to such new situations. The Babylonian Programming systems [19, 20, 18] allow programmers to annotate their source code with examples that are then used to display the results of expressions directly within the source code. Thus, they can get feedback on dynamic behavior anywhere in the environment. The Lively environments [10, 15] bring the ideas of exploratory programming to web programming by allowing programmers to develop a web application from within itself. As modern applications are often based on several languages, the Squimera and the TruffleSqueak environment support exploratory programming for such systems through polyglot exploratory tools [17, 18]. Finally, as exploration involves the creation of alternative solutions, the CoExist environment supports programmers with managing and switching between multiple variations [23].

These systems, and similar ones created by others, are designed to support exploratory practices. To make use of them, programmers need to have basic knowledge of exploratory practices, so that they can recognize how the tool supports them and to recognize situations in which the tool or environment is applicable.

We have experience in teaching exploratory programming on the undergraduate as well as the graduate level. For most of our courses we use Squeak/Smalltalk. As we teach undergraduate courses and many students continue their graduate studies at our university, we can work with the same students several times during their studies. At the undergraduate level, we teach two lectures, one on software architecture and one on software engineering, each spanning three months. During these lectures, the students work on projects in Squeak/Smalltalk. At the graduate level, we teach seminars on software design, programming tools, execution environments, and modularity. All of the seminars focus on project work in Squeak/Smalltalk or Lively. Throughout this time, we are able to observe how they acquire exploratory programming practices. While these observations are not empirically verified, they serve as a starting ground to determine which practices beginners pick up by themselves and which ones need to be taught explicitly, for example through patterns. A general observation is that students progress from learning the language Smalltalk, to learning individual tools of Squeak/Smalltalk, to making use of the whole environment, and eventually learning more general practices of exploratory programming.

2.2 A Purely Object-oriented Programming System

We describe the solution and examples of each pattern from the perspective of purely object-oriented, exploratory programming systems [29], namely Squeak/Smalltalk [11, 7]. To make the pattern descriptions accessible, we provide a short background on this perspective by briefly introducing the basic concepts.

The main element of such systems are *objects*. Objects are used to represent entities relevant to the system, for example a domain-specific entity such as a person, a system-specific entity such as a file, or basic information such as a number. In particular, an object stores the data relevant for that entity, for example a person's

name and date of birth, or a file's most recent modification timestamp. Beyond the data, an object also has behavior, which can be invoked by sending a message to the object. The sum of all behaviors of all objects defines the behavior of the system. The behavior of objects is typically the same for all objects of the same kind, for example all chat message objects can be send to a person and can create an object representing a reply message. This common behavior of one kind of object is captured in an abstraction called a *class*, for example a chat message class. All objects of one kind have the same class. Thereby, the behavior described in the class is re-used for all those objects.

Beyond this basic principle, the Squeak/Smalltalk perspective takes a different angle on the notion of systems and programs. In Smalltalk systems, the program or application to be created is part of the running programming environment. So, programmers do not create a program or system outside of the environment, but change the running environment itself from within to make it behave the way they want. As programmers can access anything in that environment, and the environment is the running program or application they want to modify, they can access all parts of the running program or application. To store such a system and share it with other, Smalltalk systems can be saved into an *image*. This feature is similar to hibernation in operating systems; all state and all running processes is saved into a file. The system can be restarted from that image file and will be in the exact same state as it was when the programmers saved it.

2.3 Pattern Audience

Our pattern collection is motivated by making exploratory programming practices learnable by novices. Thereby, programmers are users of an environment and try to employ the practices during programming. However, programmers can also be the builders of their environments. As such, they might want to make use of the patterns to get guidance in how to shape an environment for exploratory programming.

For learning the practices, the patterns are useful for exploratory programming novices and experts alike. When talking about novices, we refer to programmers new to exploratory programming. This includes programming novices, who are unfamiliar with programming in general, as well as programming experts, who are already familiar with programming, but not with exploratory programming. Both benefit from the pattern representation of practices. So far, learning exploratory programming either required a lot of time to build up personal experience, or an experienced teacher, regularly demonstrating practices by example. With the patterns, novices can now learn the practices by themselves.

Experts of exploratory programming may still benefit from the practices. As there are no written, in-detail descriptions of the practices of exploratory programming, most programmers only have their individual experience to go by. Through the patterns, they can contrast their implicit techniques with the experience of others.

Further, they may discover variations within and commonalities between practices they are not aware of, which might make their practices more effective.

Independent of a programmer's skill level, the patterns may help programmers building tools and languages fit for exploration. Tool builders may refer to the exploratory programming patterns to determine what their language or environment needs are to support certain practices. Further, they might choose to support particular practices and the patterns may provide some background on when the practice is used or how it may be altered by programmers.

2.4 Pattern Form

Since the introduction of patterns [2, 1], different communities and authors have taken up the idea and created their own pattern collections. While they all agree on the idea of a pattern as the description of the core of a solution, they differ in the form they use to describe the patterns. The main difference between these forms is the list of aspects described for each pattern.

The original pattern descriptions by Christopher Alexander consist of the name of the pattern, the context, the problem described as a set of forces, the solution, trade-offs in the solution, and a set of related patterns. The "Gang-of-Four" book, which popularized patterns in the software development community, uses a more form that includes several detailed sections describing the solution [6]. Yet another form was used in the learning and presentation patterns [9, 8], which featured a summary of the pattern consisting of only one line.

For this chapter, we use the following form:

- **Intent** is a short summary of the pattern including the fundamental challenge as well as a glimpse of the solution.
- **Motivation** describes the problem programmers might encounter during their exploration. It describes the domain and the context in which the problem occurs. This section concludes with a summary of why programmers may develop a "desire for exploration" in this situation.
- **Forces to Resolve** describes the different constraints and considerations when applying the practice. Whenever adapting the pattern to a specific situation, these forces may influence the specific adaptations.
- **(Towards a) Solution** describes the specific techniques making up the practice, including variations.
- **Consequences** describes what is required from a programming system to support this practice. It also points out technical challenges that may arise during the particular practice used in exploration.
- **Notes on Squeak/Smalltalk** illustrate how the practice would be applied in the Squeak/Smalltalk system.

3 Patterns to Enable Exploration

Our first collection of two patterns is about *enabling exploration.* In purely object-oriented systems, there is much structured information available. When investigating bugs or adding features, programmers access the object graph to understand what is there and what is missing. When explicating thoughts, programmers benefit from naming objects and then organizing those *tangible references* in spaces. Since modern programming tools offer graphical interfaces, programmers also have to make sense of a program's visual output. In sum, this combination of typing (names) and clicking (on shapes) represents an entry point to exploratory practice.

3.1 Tangible Names

Maybe also known as "Object Bindings" or "Names in Spaces."

Intent

Names help people denote accessible meaning of otherwise transient thoughts. Therefore, programmers should use names to organize not only code artifacts but all relevant objects. Programming environments should allow for flexible attachment of such names. Consequently, the use of established vocabulary should yield access to the underlying artifacts.

Motivation

In object-oriented environments, all structured information is represented as objects that have relationships to other objects. Those structures can be very deep and thus hard to follow and abstract. Code objects typically have intrinsic names to be easily identified. Many other objects, especially those that occur at run-time, may not have a (derived) textual representation that helps programmers in their understanding.

Programmers are in a constant learning process. They communicate with domain experts (or customers) to understand the rules and requirements that should be somehow represented in code. Along the way, programmers make all kinds of observations—such as computational results—that need to be documented to not get lost. Ideas emerge and become clearer. Consequently, such emergent clarity needs to be denoted *before* becoming program code, like sticky notes in the programming environment.

Programmers work with names on a regular basis because source code is filled with such textual identifiers for classes, methods, and all kinds of variables. Names help explicate thoughts; they encode meaning. In an environment where all kinds of objects can be materialized thoughts, names play an important role in keeping track.

The question is whether programmers are willing to write down and handle names when talking (and reasoning) *about* programs and their informational trails.

Today's programming tools are full of textual labels. There are code browsers or object inspectors, which employ text fields or lists with labels. Programmers rely on their recognition of an object's intrinsic names to look up and find information of interest. Also, text-based search is a common entry point in program understanding. Programmers just type (part of a) name into a text field and expect interesting objects to show up in a (text-based) result list. Name it, spell it, type it, find it.

Programmers' Desire for Exploration. An object's inherent structure does not yield a name appropriate for the current task. The programmer wants to reduce cognitive load by explicating and working with new names in the environment:

- Attach a name to an object for later reference.
- Look up the object structure for any name that is visible on screen.
- Share names between several tools (or scopes).
- Organize thoughts on different levels such as domain, task, or personal.

Such names may change. They can be mere nicknames (or mnemonics) in the beginning.

Forces to Resolve

Programmers usually understand the importance of good names in source code, but they might hesitate to bring the same attention to names that appear in the entire programming environment:

- Names may not be reachable outside a certain tool or other scope.
- In the "offline" world, taking notes is *very* easy.
- Arbitrary name lookup is not possible for arbitrary labels in (graphical) tools.
- A good name is hard to find.
- Recognizing a name on screen "feels good" and reduces cognitive load.
- Extra references to objects consume extra resources in the environment.
- The same name can change meaning over time.

(Toward a) Solution

Programmers have to come up with and refer to names all the time when writing or reading source code. This very habit (or custom) is the base for working with *tangible names* during exploration. Names can be very helpful to organize objects, even outside a program's code base.

Know about and rely on the system's vocabulary. At the beginning of an exploratory session, programmers work with the names that already exist and are accessible for technical artifacts. For example, there are package names, class names, or method names. Besides such language artifacts, there can be names for run-time objects such as the current process and active method. The environment should make this basic level accessible through names to serve as entry points for exploration. The programmers' thoughts may spin around those technical artifacts, which triggers the desire to learn more about what's going on.

Just write down that name. Once an idea or observation starts to gain clarity, programmers will try to describe it to explicate meaning—or at least give it a token to further think about it, to not forget about it. In the programming environment, programmers should be able to just type names for further reference. There might not even be an object attached to such names yet. Still, programmers can now go looking for objects that deserve such names.

Attach (more appropriate) names to objects. Many programming tools display characteristic object structure in textual form. Given a programmer's current task, a name may come up that would serve as a more appropriate identifier. Programmers should attach such a name to the particular object so that its meaning can be recalled more easily. Programming environments should allow for adding any number of extra names to objects.

Organize names in spaces. Programmers should find "an empty sheet of paper" or "a clean whiteboard" to document their thoughts in the exploration process with little friction loss. For example, windows with big empty text fields are common metaphor to represent such spaces on screen. Such spaces can directly represent task scopes; they may be even expanded to document overall domain knowledge. Programmers can collect names and attach them to objects on the fly. Like sticky notes on whiteboards, names can be moved around to influence each other in the overall process of program understanding.

Resolve names to reveal structure. Within a certain space, programmers should be able to directly resolve the names they have just typed or observed in a tool's graphical display. The environment should keep track of the names' connection to the underlying object and hence the structured information. As an effect, the object's (intrinsic) textual form can appear or a more sophisticated tool can offer a means to explore structure. In text widgets, such name lookup resembles code evaluation. In list widgets, the connection of any visual label to an underlying model (and thus object) might be more challenging if not supported by the tool framework.

Combine spaces to integrate exploration paths. Programmers should reflect on the spaces they currently use for name collection. Related themes may emerge, which requires to combine spaces (or at least bring names over from one space to another.) Programmers should avoid connecting "loose ends" in offline notes. Instead, they should employ the means in the (digital) environment such as shared clipboards or drag-and-drop gestures. Consequently, the environment should offer a basic model for tool (and thus name) integration without compromising data quality.

Dismiss the spaces you no longer need. Programmers should reflect on their current task's progress. Once finished, spaces should be dismissed. Adding extra

names to objects may interfere with the environment's automatic clean-up mechanism. Since resources are usually limited, discarding names (or entire spaces) is part of the exploratory process. Note that there are usually means to recover from mistakes. Some environments may offer automatic dismissal of no longer needed spaces, which programmers may have to configure to accommodate their working habits.

Consequences

Being able to name objects requires object representations for information in the environment. External data can usually be imported as generic structures such as maps and dictionaries. Materializing low-level language (and run-time) concepts, however, can be more challenging for the provider of the programming system. Yet, programmers are likely to include "behavior" or "execution stack" in their thoughts when thinking about specification and implementation.

Having the freedom of reasoning about any accessible object with new names in custom spaces, programmers can easily break abstractions. There are environments without a certain compilation boundary, that is, source code access to all parts in a system, which demands a certain discipline for information hiding. Programmers need to be aware of not "leaking" usually hidden information into new source code, that is, after the exploratory session.

Aliasing is already a challenge in object-oriented architectures. During exploration, programmers add even more names to the same objects, which makes identity a rather intangible, hardly explicable concept. As different (work)spaces support overlapping names, tools for overview might mitigate this consequence. On the other hand, working with external data (and distributed structures in general) implies a comparable challenge outside the context of exploratory programming practice.

Names can point to outdated structure without programmers' being aware of it. Programs (under observation) only manage *their* point of view. Programmers hold on to certain objects by chance, but have often no means to notice when related objects "lose interest" in their direct neighbors.

Notes on Squeak/Smalltalk

In Squeak/Smalltalk, programmers can write notes into workspaces, which are interactive text buffers that support code evaluation like a read-eval-print loop (or REPL). The Smalltalk language can be used as a scripting language in almost any other tool's text fields to set up new, but tool-local, name bindings. The combination of such tool spaces is possible, for example, through global variables. There are globals (and reserved keywords) that reference basic run-time information such as `thisContext` for the active method (context) and `ActiveWorld` for the topmost GUI object.

Many graphical tools in Squeak retain a (more or less direct) connection between visual label and underlying object. This connection allows programmers to explore

underlying structure through simple pop-up menus or drag-and-drop gestures. Vivide [26, 25, 24] is a tool-construction framework on top of Squeak/Morph that preserves such a direct connection in the GUI by design. Consequently, programmers can resolve names not only in text fields but other interactive widgets, too.

3.2 Tangible Pixels

Maybe also known as "Meta Menu" or "Shape Halo" or (more generic) "Direct Manipulation Interface."

Intent

Visual shapes can raise attention and trigger curiosity to explore. Programmers use interactive spaces to organize graphical representations on screen. Programming environments should allow to "look behind" visual shapes to explore the underlying objects and relationships. In practice, programmers can point and click to manipulate such shapes directly.

Motivation

Using high-resolution, graphical displays, programmers can create convincing illusions of tangibility merely through colorful pixels on a two-dimensional plane. Even if a program-under-construction has no elaborate visuals itself, today's programming tools (and environments) can offer visualization to clarify (code) structure. The question is whether programmers accept graphical interfaces only from a user's perspective or whether they also try to work with *visual shapes* as a tangible medium under construction.

The shared (programming) environment uses objects to represent everything, including graphical primitives. There can easily be extra gateways to connect "what is visible" to "what it is made of." Sometimes, a widget's affordances guide programmers to shorten the feedback loop in their exploratory journeys—such as clicking a button nearby to reveal a pop-up menu. Yet, extra (hidden) gestures may have to be learned to enable exploration.

While the connection between a visual shape to *any* underlying object may be simple, finding *useful* paths to descriptive model data may not be. That is, *spatial distance* can be reduced with elegant software design, while *semantic or temporal distance* often remains part of the exploration efforts.

Programmers' Desire for Exploration. A visual shape on screen makes the programmer curious because it may indicate a bug or place for a new feature. The programmer wants to reduce cognitive load by directly navigating from the pixels to objects and hence structured information:

- Understand the structure behind flat pixels.
- Open tools to explore that structure, to make it tangible.
- Keep the connection between tools and visuals on screen.
- Organize thoughts on different levels such as domain, task, or personal.

Direct manipulation (for exploration) helps shorten the feedback loop.

Forces to Resolve

Programmers usually design graphical interfaces for usage only, not for exploratory (debugging) practice. Still wanting to understand how the underlying objects enable the program's purpose, programmers might hesitate to even try using the same interface to also "look behind the curtains," that is, the visual shapes. The following forces emerge:

- The program's GUI has no extra code to enable debugging.
- Visual objects should directly relate to model data in the domain.
- The scene graph is too deep and complex.
- The visuals are too small to point at.
- There are no distinct, steady shapes; it is more like animation.

(Toward a) Solution

Graphical output is often the "result" of the system. Starting the exploration from there means programmers start from something that they can grasp and that is already tied to a purpose.

Point and hover. Many visual shapes on screen offer extra information when programmers hover the mouse cursor over them and wait for a bit. Then, descriptive tooltips (or balloon texts) appear as overlays nearby. Such user interaction strengthens the blending of pixels into tangible compounds (or graphical objects). In programming tools, the revealed insight can indeed help programmers to look at object structure. In other (regular) programs, such information might be targeted toward its users, not programmers who what to "look behind the curtain."

Look for and click on meta buttons. There are UI elements that do not invoke immediate side effects on the system (or program). Such elements are often clickable buttons that offer possible actions through pop-up menus. Looking at such actions, programmers can get a better understanding of what object is actually displayed in

pixels nearby. Similar to hover effects, the emerging visual compounds support the connection between pixels and underlying structure.

Employ reserved (meta) input gestures. Three-button mice render click-on instructions ambiguous. The primary click on, for example, buttons or list elements is part of the common bi-manual interaction mode—keyboard plus mouse. Yet, there can be many other input gestures (such as keyboard shortcuts) that encode special modes or means to interact with visual shapes. Users may want to *talk about* a thing on screen when they perform a secondary click and expect a pop-up menu to show up. Programmers should know such meta gestures as they might exploit underlying objects. There are environments that make the entire scene graph tangible.

Enter gateways to reach (meta) tools. Programmers should follow the shortest path available to explore the connection between the visuals on screen and the underlying object structure. Programming environments should allow for such short paths through reserved input gestures. That is, there should be a connection between a program's run-time objects and the objects that make up the source code (or other resources).

Organize programs and tools in spaces. When the environment offers the means to explore running programs and structure-revealing tools side by side, programmers should organize those in (visual) spaces. Such spaces help document exploration paths (and overall progress) in a tangible way.

Consequences

The visual design may be in conflict with serving both user and programmer. Allocating extra screen space for buttons (or similar) might confuse users, which would defeat the primary purpose of that program. Also, increasing shapes' sizes so that programmers can click on and "look behind" the surface might not be a viable option either.

Extra input gestures—dedicated to exploratory programming—would not be useful for regular users. Already, there is often a dispute on supporting common keyboard shortcuts for common (user) operations. Mouse buttons are limited and so are keys on the keyboard. Taking away more options would interfere with this discussion from a new perspective.

Programmers would have to learn about extra interface elements and how to use them while running the program under construction. It can already be challenging to organize non-visual objects, and separate essential from supportive. Visual objects further aggravate this issue. User interface and programming interface might blend, which could be okay for programmers, but frustrating for users.

Depending on the system's rendering pipeline, preserving a pixel-to-object mapping can be challenging. If not supported by the underlying graphics framework by design, extra programming effort may be required to at least offer such a connection for selected programs.

Notes on Squeak/Smalltalk

Squeak has always supported one-button mice in making point-and-click interfaces discoverable and simple to use. For example, there is a button for a list widget's menu, placed in the scrollbar. There is no need to learn a secondary click: rather the user first clicks on a list element, then clicks on the menu button to show available actions for that element. Note that three-button mice are also supported. In that case, the secondary click avoids extra mouse movement.

In Squeak/Morphic, all graphical objects—so-called morphs—can be selected through a special gesture. Then, a "context menu" appears in the form of a halo around that object. While this menu can be used as part of the regular user interface, it also offers a gateway to programming tools such as object inspectors and code browsers. The halo concept originates in the *outliner* in the Self system [28].

The collection of pixels that represent a graphical object can be difficult to see. The Morphic halo appears as a rectangular "outline", which can be invoked whenever the programmer has a reference to such a graphical object (or morph). Consequently, there is also a direct connection from object to pixels—not just the other way around.

4 Patterns to Control Exploration

Our second collection of two patterns is about *controlling exploration.* Programmers have to trust their programming environment and tools. While learning about a problem domain, implementation strategies, and personal preferences, programmers will gain trust in their tools if those can mirror that progress. If one's mindset can be observed on the screen, programmers will get a feeling of being in control. First, they can set up boundaries to avoid making mistakes and derailing, but staying focused instead. Second, they can establish an area to safely work within, which includes reliable recovery and cleaning up after the exploration task.

4.1 Configurable Constraints

Maybe also known as "Configurable Guides" or (more generic) "Domain-specific Environments."

Intent

"With great power comes great responsibility." Being deep in an exploration activity, programmers benefit from meaningful limitations while they progress—to stay in focus and avoid mistakes. Programming environments should allow for configurations that constrain or guide the tangible notion of names and pixels.

Motivation

To foster the programmer's mindset for exploration, the environment should take care of traps that would otherwise distract or intimidate. When programmers have to fear drastic consequences, they might resort to unchecked hypotheses instead of exploring and learning about what is really happening. In self-sustaining systems, such consequences could entail broken tools or lost data, which in turn means higher costs in the software development process.

Distraction may come from standard tools showing irrelevant information such as low-level code in debuggers or irrelevant modules in browsers. Intimidation may come from a sheer overwhelming amount of possibilities. Luckily, many tools can be tailored to specific exploration strategies. Programmers can reduce cognitive effort when screen contents match their mental model as closely as possible.

Domain-specific tools can help guide programmers actions in a generic fashion. That is, a tool's specificity can complement its expressiveness. Programmers should always stay in control; they decide how they want to proceed. Tools (and environments), however, help programmers remember and apply best practices.

Programmers' Desire for Exploration. Programming environments offer many complementary tools. Programmers have to explore their choice of tools as well as the information visible through these tools:

- Keep going and stay in focus.
- Explore *relevant* object structure.
- Hide *irrelevant* implementation details.
- Use non-limiting support for the current task.

Meaningful constraints can promote a state of "flow" [3] in exploratory programming.

Forces to Resolve

When not following *a single plan* but exploring possibilities to gain understanding, programmers may hesitate to freely embrace exploration within the programming system:

- It is hard to recover therefore mistakes must be avoided.
- It is hard to focus because generic programming tools "leak" implementation details.
- It is hard to proceed because domain-specific tools impede general-purpose programming if needed.

(Toward a) Solution

Programmers can avoid many mistakes and stay focused within exploration by mastering the means of representing information in the environment. That is, they have to choose the right tools, tweak tool parameters, and know when to change plans.

Choose tools appropriate for exploration. Programming environments usually have many different tools for many different programming tasks. There is often no "one fits all" solution; a notable overlap in tool features can occur. Programmers should choose from tools that fit the desired exploration strategy. Selection criteria include accessibility and representation of relevant software artifacts.

Configure to accommodate specific needs. Programming tools are often "general purpose" but also offer configuration parameters to accommodate specific domains, tasks, or personal preferences. Therefore, programmers should schedule extra time to tweak those parameters. Especially at the beginning of exploration, known characteristics of the current problem domain can already be included.

Reflect and realize when to change strategies. Being deep within an exploration path, programmers should account for extra time to reflect on the current working mode. Different tools might be more appropriate to continue, including generic code browsers. Different configurations of the tools in use might yield more promising results. That is, exploring the problem and solution spaces includes exploring the available means to do so.

Start exploration in empty spaces. Programmers should avoid interfering with the results of other work in the system. A new exploration (path) should start in a rather empty space such as a new instance of a tool window. Programming environments should account for having enough space to follow many different hunches.

Migrate progress to new tools. When programmers choose to switch tools, they should also try to bring existing insights along. That is, all names and meaningful objects, including visuals, should remain (somewhat) accessible in the other tool's interface. There will be compromise because different tools have different strengths and levels of data support.

Consequences

Tool configuration may blend into tool construction, which may take unexpected time and effort. Especially in open systems where programmers can access and modify the entire codebase, one has to carefully "timebox" any attempt to change the status quo. Thus, the matter of "staying focused" becomes double-edged: using tools and also configuring them.

Switching tools *and* transferring (intermediate) results may only work within a certain environmental boundary. If the underlying representation of structured information differs fundamentally, programmers might have to compromise and serialize parts of this information as they see fit. If such a reduction in quality is not an option, programmers can try to integrate external tools directly into the programming environment.

Trial-and-error remains part of the exploration process. Programmers cannot always know when to switch tools. In any case, the overall programming task may still be "timeboxed," leaving only limited resources for out-of-plan exploration. However, such a limitation can be an obvious trigger for programmers to "just try something different" in any remaining period.

Notes on Squeak/Smalltalk

Squeak comes with tools that are tailored to the Smalltalk language. Class browsers show source code; object inspectors show instance variables; debuggers accurately display the context of method activations. Consequently, guidance comes from the (hopefully descriptive) names of code artifacts. While there are simple filters, programmers have to selectively disregard unrelated information. There are no on-board means for higher-level, domain-specific perspectives that could guide exploration.

Luckily, there are frameworks and libraries that build on top of Squeak/Morphic, which programmers can install to support exploration. These includes projects that aim to improve programming education and programming experience in general. Yet, they can play part of their role in (general purpose) exploratory practice. Etoys [5] and Scratch [16], for example, both hide textual code complexity through visual shapes—meant to be composed and explored through click, drag, and drop. Then, there is Babylonian programming [21, 19, 18], which embeds concrete values into abstract code so that programmers do not have to stray and lose time in breakpoint-triggered debuggers. There are also object-focused, script-based means to construct new tools for exploration with the Vivide framework [26, 25, 24]. Programmers therefore have many alternatives to choose from.

4.2 Reliable Recovery

Maybe also known as "Safety Net" or "Back to the Start" or "Checkpoints."

Intent

Programmers leave traces during exploration. Those traces may need to be altered when backtracking or removed when finishing. Programming environments should allow for configurations that manage (or constrain) side effects on software artifacts (including the tangible notion of names and pixels).

Motivation

Programmers consume many different kinds of information when trying to understand programs and possibilities. Yet, consumption can entail change such as disassembling a closed box. It is thus advisable to take extra care to scope the effects of such exploration. That is, programmer's do not just observe, but they actually "poke around" to learn how specific objects react.

The most obvious solution—known from "traditional" programming practice—is typically too costly: throw away everything and start over. There can be a non-deterministic state, which is hard to replicate for another round of exploration. When programmers are continuously modeling artifacts in a running system, restarting might also imply tediously retyping source code or remodeling other essential resources. Luckily, there have been approaches that shorten the cycle of recovery to try again or continue work.

Programmers' Desire for Exploration. Programming environments can be both messy and tidy at the same time. It is very easy to create empty spaces; it is "just" digital software. Programmers want to dive into the exploration task:

- Keep going and stay in focus.
- Backtrack when hitting a dead end.
- Clean up when finished exploring.
- Quickly recover when having broken something by accident.

Programmers can easily forget about that clean-up, which can later become a reason for unnecessary recovery.

Forces to Resolve

Having the system's state made of interconnected objects, programmers have to take care of those objects and their relationships during exploration. Like cleaning up your study may be not worth the effort, programmers may hesitate to follow exploratory practices:

- Living with "brittle" (run-time) state around for too long feeds the urge to "reboot" and start afresh.
- It is hard to disseminate the "broken" from the useful state.
- It is costly to throw "everything" away.
- It is hard to anticipate the effects of exploration tools (and actions) upfront.

(Toward a) Solution

Even within a constrained and guided setup, programmers can make mistakes and need to recover. A system's object graph may just be too complex to foresee the effects of every possible action.

Use tangible (and easily discardable) scopes. During exploration, programmers grow a collection of (perhaps newly) named objects. This collection should represent a scope that can easily be dismissed when finished. The environment's resources are typically limited; automatic clean-up works only through computational, user-independent rules. Thus, programmers must explicitly indicate the state of exploration as they see fit. A tangible scope can be pointed to, and thus helps with such indication.

Establish distinct steps on a path. Programmers should modularize their exploration path. At best, an obvious (only linearly dependent) sequence of steps (or tools or scripts) can be re-evaluated repeatedly while the program (under observation) keeps running. Along such paths, programmers can easily backtrack and revise their choices.

Create checkpoints for safe retreat. Programmers should replicate (or copy) a specific setting before experimenting with unknown side effects. This has a similar effect to the way that children can have repeated fun by coloring (by numbers) on a photocopy, rather than on the original. Programming environments should offer clear guidelines to specify and duplicate (part of) the object graph. Clear boundaries, like shielded sandboxes, can further help to establish trust between programmers and their environment.

Hit the pause button to take a break. Exploration can be time-consuming. Programmers have to consider their working schedule and thus maybe interrupt a session. Thus, programming environments should offer a means to pause all action in the running system—or selected modules. On the one hand, programmers can then take a closer look at such "snapshot of time" to better understand the objects and messages *in situ.* On the other hand, programmers can actually take a break and rely on the system to continue running—exactly where it left off—the next day.

Consequences

Modularity in the exploration path largely depends on guides and constraints offered through tools and their interfaces. If programmers would be forced to put much effort into refactoring existing steps, chances are that they would not do it. Such extra effort would interfere with their focus and thus interrupt the "flow." Consequently, the modular description of exploration steps is one of the primary challenges in domain-specific tool construction.

At the same time, there can be *too many* checkpoints, outliving past exploration tasks and demanding extra resources. Programmers might hesitate to discard (even tangible) scopes because these form new objects of value, that is, documentation for later use. There can always be new but similar challenges in the near future; one

cannot know upfront. Yet, the actual value of such (maybe outdated) checkpoints can be difficult to assess, even in retrospect.

Programmers might avoid creating *complete* checkpoints for reasons of cost. It might even be impossible to strive for completeness. External resources can be especially difficult to grasp; stubbing them can interfere with trust in the exploration's outcome. In other words, working with *real data* is a problem force that is not addressed through this pattern.

Notes on Squeak/Smalltalk

Squeak's tools (and associated windows) can represent tangible scopes to organize exploration and clean up after it. For example, programmers close *workspace windows* to dismiss bindings and thus *tangible names*. They also organize multiple windows in *projects* (or "desktops"), which can easily be closed to dismiss open tools and thus *tangible pixels*.

Within a single workspace (window), programmers modularize (partial) scripts through text lines of source code. Consequently, they are in charge of orchestrating simple inspection or effectual experimentation. At best, programmers can re-evaluate the entire code in a workspace without breaking things or "polluting" the environment with useless data.

Programmers can hit the key combination `[CMD]+[.]` at any point to suspend the currently running process. That is, they can pause message passing for a specific portion in the system, usually the UI process. After inspection, suspended processes can then be resumed—or terminated to free resources. In combination with Squeak's image, programmers are basically in control of (execution) time. Yet, there is ongoing research on how to offer more elaborate tools for immediate recovery in Squeak. For example, CoExist [23] offers fine-granular revisions for code changes without needing programmers' anticipation of mistakes.

5 Discussion and Future Work

A pattern's success is measured through relevance, quality, and impact. Its mere discovery is of less importance. In this chapter, we attempt to describe aspects of exploratory practice in pattern form for the first time. The result is a collection of "drafts" that need to be polished and revised. Yet, the sole artifact of a pattern is not useful unless applied in practice. That is, fellow programmers who use the practice of exploratory programming should see value in our work. Therefore, like many patterns and pattern-authors before us, we seek feedback from both practitioners in the field and the pattern community—which takes time and several pattern-writing workshops.

The patterns we drafted are very broad and leave many questions unanswered. We made an attempt to formulate not only actions for programmers but also advice

for tool builders. From experience we know that both tool usage and construction go hand in hand. Indeed, it may be the same programmer who switches between roles many times during the same programming task. Consequently, our patterns can reveal shortcomings in programming environments regarding its tools and means for construction. Since time is always a scarce resource in software development, some patterns may thus not be feasible to apply. We want to address such situations in our next revisions in also offering more paths to enable exploration.

In our next steps, we will tackle verbosity to make each pattern's intent more clear. Especially the solutions we propose in each pattern are likely to be split up into patterns of their own, leaving the current form as possible categories for orientation. Of course, when discovering complementary patterns or new perspectives as a whole, the entire organization can change. In the process, we will also investigate practices beyond object-oriented systems, because exploration happens in every programming environment. Our vision is to collect and materialize an accessible catalog of patterns—maybe even create a pattern language [2]—that can serve as a reliable reference in daily programming practice.

6 Conclusion

In this chapter, we described typical approaches of exploratory programming practices as they occur in education and research through the Squeak/Smalltalk system. We gained many of our own experiences with this system's concepts, which already originated in the 70s and hold up splendidly for today's challenges. The system's purely object-oriented design offers many interesting perspectives on program understanding and debugging with short feedback loops. First, we covered patterns to *enable* exploration, which unpacks the role of textual labels and visual shapes. Second, we addressed patterns to *control* exploration, which emphasizes not only avoiding mistakes but also embracing them through trusted means for recovery.

This chapter is only the first step toward a more substantial collection of patterns, maybe a whole pattern language, that can serve programmers in many domains. Even at this early stage, we imply many valuable aspects of exploratory programming practice to be further unpacked in pattern form. We believe that such a comprehensive, accessible catalog can help connect many overlapping efforts in contemporary programming language and tool research.

Acknowledgements We gratefully acknowledge the financial support of the HPI Research School on Service-oriented Systems Engineering (`www.hpi.de/en/research/research-schools`) and the Hasso Plattner Design Thinking Research Program (`www.hpi.de/en/dtrp`).

References

1. Alexander, C.: The Timeless Way of Building. Oxford University Press (1979)
2. Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S.: A Pattern Language - Towns, Buildings, Construction. Oxford University Press (1977)
3. Csikszentmihalyi, M.: Flow: The Psychology of Optimal Experience. Harper Perennial Modern Classics (2008)
4. DeGrace, P., Stahl, L.: Wicked Problems, Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms. Yourdon Press computing series. Yourdon Press (1990). URL https://books.google.de/books?id=__omAAAAMAAJ
5. Freudenberg, B., Ohshima, Y., Wallace, S.: Etoys for one laptop per child. In: 2009 Seventh International Conference on Creating, Connecting and Collaborating through Computing, pp. 57–64. IEEE (2009)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., USA (1995)
7. Goldberg, A., Robson, D.: Smalltalk-80: The Language and Its Implementation. Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, USA (1983)
8. Iba, T.: Presentation patterns: a pattern language for creative presentations. Lulu. com (2014)
9. Iba, T., Sakamoto, M.: Learning patterns III: a pattern language for creative learning. In: L.B. Hvatum (ed.) Proceedings of the 18th Conference on Pattern Languages of Programs, PLoP 2011, Portland, Oregon, USA, October 21-23, 2011, pp. 29:1–29:8. ACM (2011). DOI 10.1145/2578903.2579166. URL https://doi.org/10.1145/2578903.2579166
10. Ingalls, D., Felgentreff, T., Hirschfeld, R., Krahn, R., Lincke, J., Röder, M., Taivalsaari, A., Mikkonen, T.: A world of active objects for work and play: the first ten years of lively. In: E. Visser, E.R. Murphy-Hill, C. Lopes (eds.) 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016, Amsterdam, The Netherlands, November 2-4, 2016, pp. 238–249. ACM (2016). DOI 10.1145/2986012.2986029. URL https://doi.org/10.1145/2986012.2986029
11. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: The story of squeak, a practical smalltalk written in itself. In: Proceedings of OOPSLA 1997, vol. 32, pp. 318–326. ACM (1997). DOI 10.1145/263698.263754. URL http://doi.acm.org/10.1145/263698.263754
12. Kay, A., Goldberg, A.: Personal dynamic media. Computer **10**(3), 31–41 (1977)
13. Kery, M.B., Myers, B.A.: Exploring exploratory programming. In: A.Z. Henley, P. Rogers, A. Sarma (eds.) 2017 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2017, Raleigh, NC, USA, October 11-14, 2017, pp. 25–29. IEEE Computer Society (2017). DOI 10.1109/VLHCC.2017.8103446. URL https://doi.org/10.1109/VLHCC.2017.8103446
14. Lincke, J., Krahn, R., Ingalls, D., Röder, M., Hirschfeld, R.: The lively partsbin-a cloud-based repository for collaborative development of active web content. In: 45th Hawaii International International Conference on Systems Science (HICSS-45 2012), Proceedings, 4-7 January 2012, Grand Wailea, Maui, HI, USA, pp. 693–701. IEEE Computer Society (2012). DOI 10.1109/HICSS.2012.42. URL https://doi.org/10.1109/HICSS.2012.42
15. Lincke, J., Rein, P., Ramson, S., Hirschfeld, R., Taeumel, M., Felgentreff, T.: Designing a live development experience for web-components. In: L. Church, R.P. Gabriel, R. Hirschfeld, H. Masuhara (eds.) Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience, PX/17.2, Vancouver, BC, Canada, October 23-27, 2017, pp. 28–35. ACM (2017). URL https://dl.acm.org/citation.cfm?id=3167109
16. Maloney, J., Resnick, M., Rusk, N., Silverman, B., Eastmond, E.: The scratch programming language and environment. ACM Transactions on Computing Education (TOCE) **10**(4), 1–15 (2010)
17. Niephaus, F., Felgentreff, T., Pape, T., Hirschfeld, R., Taeumel, M.: Live multi-language development and runtime environments. The Art, Science, and Engineering of Programming **2**(3) (2018). DOI 10.22152/programming-journal.org/2018/2/8. URL http://dx.doi.org/10.22152/programming-journal.org/2018/2/8

18. Niephaus, F., Rein, P., Edding, J., Hering, J., König, B., Opahle, K., Scordialo, N., Hirschfeld, R.: Example-based live programming for everyone: Building language-agnostic tools for live programming with lsp and graalvm. In: Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2020, p. 1–17. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3426428.3426919. URL https://doi.org/10.1145/3426428.3426919
19. Rauch, D., Rein, P., Ramson, S., Lincke, J., Hirschfeld, R.: Babylonian-style programming - design and implementation of an integration of live examples into general-purpose source code. Programming Journal **3**(3), 9 (2019). DOI 10.22152/programming-journal.org/2019/3/9. URL https://doi.org/10.22152/programming-journal.org/2019/3/9
20. Rein, P., Lincke, J., Ramson, S., Mattis, T., Niephaus, F., Hirschfeld, R.: Implementing babylonian/s by putting examples into contexts: Tracing instrumentation for example-based live programming as a use case for context-oriented programming. In: Proceedings of the Workshop on Context-oriented Programming, pp. 17–23 (2019)
21. Rein, P., Ramson, S., Lincke, J., Hirschfeld, R., Pape, T.: Exploratory and live, programming and coding - A literature study comparing perspectives on liveness. Art Sci. Eng. Program. **3**(1), 1 (2019). DOI 10.22152/programming-journal.org/2019/3/1. URL https://doi.org/10.22152/programming-journal.org/2019/3/1
22. Sheil, B.: Power tools for programmers. Datamation Magazine (1983)
23. Steinert, B., Cassou, D., Hirschfeld, R.: Coexist: overcoming aversion to change. In: A. Warth (ed.) Proceedings of the 8th Symposium on Dynamic Languages, DLS '12, Tucson, AZ, USA, October 22, 2012, pp. 107–118. ACM (2012). DOI 10.1145/2384577.2384591. URL https://doi.org/10.1145/2384577.2384591
24. Taeumel, M.: Data-driven tool construction in exploratory programming environments. Ph.D. thesis, University of Potsdam, Digital Engineering Faculty, Hasso Plattner Institute (2020). DOI 10.25932/publishup-44428. URL https://doi.org/10.25932/publishup-44428
25. Taeumel, M., Perscheid, M., Steinert, B., Lincke, J., Hirschfeld, R.: Interleaving of modification and use in data-driven tool development. In: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2014, pp. 185–200. ACM, New York, NY, USA (2014). DOI 10.1145/2661136.2661150. URL http://doi.acm.org/10.1145/2661136.2661150
26. Taeumel, M., Steinert, B., Hirschfeld, R.: The VIVIDE programming environment: connecting run-time information with programmers' system knowledge. In: G.T. Leavens, J. Edwards (eds.) ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2012, part of SPLASH '12, Tucson, AZ, USA, October 21-26, 2012, pp. 117–126. ACM (2012). DOI 10.1145/2384592.2384604. URL https://doi.org/10.1145/2384592.2384604
27. Trenouth, J.: A survey of exploratory software development. The Computer Journal **34**(2), 153–163 (1991). DOI 10.1093/comjnl/34.2.153
28. Ungar, D., Smith, R.: Self. In: Proceedings of the Conference on History of Programming Languages (HOPL) 2007, HOPL III, pp. 9 – 1. ACM, New York, NY, USA (2007). DOI 10.1145/1238844.1238853. URL http://doi.acm.org/10.1145/1238844.1238853
29. Wegner, P.: Concepts and paradigms of object-oriented programming. OOPS Messenger **1**(1), 7–87 (1990). DOI 10.1145/382192.383004. URL https://doi.org/10.1145/382192.383004